X68k

Programming Series

村上敬一郎＋大西恵司＋萩野祐二……………………共著

(#2)

X680x0 libc

Vol. 2

Programmer's Reference

Manual Books

SOFT BANK

X68k

Programming Series

村上敬一郎＋萩野祐二＋大西恵司…………共著

(#2)

X680x0 libc

Vol. 2

Programmer's Reference

*LIBC*は **X68000** および **X68030** 上の **Human68k** ver.2，**Human68k** ver.3 上で動作しますが，**GCC** あるいは *XC*がすでに動作していることが必要です。したがって，**GCC** を利用する場合には最低限 2M バイトのメモリが必要です。

本書は，「*X680x0 Develop. & libc II*」の内容に即して「*X680x0 libc*」を加筆修正したものです。

バージョンアップによって仕様変更されたライブラリ関数については，新規に作成した「変更」項目にその内容が解説してあります。

- システム名，CPU 名などは一般に各社の登録商標です。本文中では，とくに TM，Ⓡは明記しておりません。

X68k Programing Series #2

X680x0 libc

Vol. 2

Programmer's Reference

C O N T E N T S

Chapter 1

Chapter 2

Chapter 4

Chapter 5

Chapter 6

A …………ライブラリリファレンスについて

Vol. 2では，ライブラリに含まれる各関数のリファレンスマニュアルを掲載します。マニュアルはおおむね次のような形式で記述されていますので，よく読んでからお使いください。なお，非常に関連の深い関数同士は，まとめて1つのページに記述されている場合があります。

用　途── 関数の用途について簡単にまとめて記述してあります。何に使うかを知るためには最も有効な項目です。

書　式── 関数を使用するために必要なインクルードファイルと関数のプロトタイプについて記述してあります。目的の関数を使用したい場合には，必ずここに指定されたインクルードファイルを読み込むようにしなければなりません。

解　説── 関数の引数の意味や関数の用法，細部の仕様などについて記述してあります。関数を使用する前によく読んでおいてください。***LIBC***は***XC***とは動作が異なるものも多く，またいろいろな拡張がほどこされていますから，従来の発想で利用すると正しく動作しない場合があります。

戻り値── もし関数に戻り値があるならば，それについて記述してあります。また関数がエラーを返すならば，そのことについても触れています。

注　意── 関数を使用する際に気をつけなければならないことがらや制限事項について記述してあります。ここに書いてあることは必ず守らなければなりません。注意してください。

互換性── ***XC***や***MS-C 7.0***などの他のコンパイラのライブラリとの互換性について，互換性が「ない」場合にはそれに関して記述してあります。これ以外にも細部で異なっている場合はありますが，それらはおおむね上位コンパチブルと考えていただいて結構です。ただし，この欄に記述があるものは互換性がないものと思ってください。

規　格── 関数を作成するにあたって参照した各種の規格，コンパイラなどの名称について記述してあります。関数の他の処理系への移植性を示していると考えてください。ただし，仕様などが一部異なる場合もあるので，目安程度に考えてください。

関連項目―― 当該関数に関連する他の関数について記述してあります。当該関数の説明だけでは不十分な場合には，これらの関連する項目についても調べてみてください。

変　　更―― 従来の関数仕様 (*LIBC* Ver.1.0.20) と「*X680x0 Develop. & libc II*」の関数仕様 (*LIBC* Ver.1.1.31) の違いについてまとめてあります。とくに変更がない場合は，記述してありません。その場合はおおむね互換であると考えてください。

なお，マニュアル中では引数をイタリック体 (*italic*) で，それ以外の関数名や変数名などをタイプフェース体 (`typeface`) というようにフォントを変えて示してあります。ただし，書式内での引数はタイプフェース体で示しています。

Chapter 1

C 標準関数ライブラリ

本章には C の標準関数ライブラリのマニュアルを掲載します。*LIBC*では，*ANSI C*で規定された *ANSI C*標準関数に加え，低水準 I/O コントロール，プロセス管理やユーザ管理など，必須の関数群をサポートしてあります。

なかには UNIX の機能をエミュレートするための関数も少なくありません。必ずしも初心者にとっては扱いやすい関数ばかりではありませんが，よくマニュアルを読んで使用してください。

_addlastsep

用　途——パス名の最後にパス区切り記号を付加する。

書　式——#include <sys/xglob.h>

```
char *_addlastsep (char *path);
```

解　説——_addlastsep 関数は，*path* で指定されたパス名の最後に**パス区切り記号**を付加する。ただし，すでに区切り記号が存在する場合は何も行わない。

戻り値——*path* を返す。

注　意——*LIBC* ではパス区切り記号は "/" と "\" の両方を認識するが，結果はすべてスラッシュ "/" になるように変換される。_addlastsep 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——_dellastsep，_tobslash，_toslash

abort

用　　途——カレントプロセスを異常終了させる。

書　　式——

```
#include <stdlib.h>
void __volatile abort (void);
```

解　　説——abort 関数はカレントプロセスを異常終了させるため，自プロセスに対して SIGABRT シグナルを送る。もし SIGABRT がキャッチされなかったり，キャッチされてもシグナルハンドラから戻ってくる場合には，プロセスは SIGABRT シグナルのデフォルトの動作を行う。すなわち，すべてのファイルストリーム，ファイルハンドルはクローズされ，バッファリングされていたデータは書き出される。シグナルハンドラから戻ってこない場合も，やはり結果的にプロセスは終了する。

戻 り 値——なし。abort 関数は決して戻ってこない。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——exit，kill，raise，signal

abs

用　　途——int 型の値の絶対値を取得する。

書　　式——#include <stdlib.h>

```
int abs (int n);
```

解　　説——abs 関数は *n* の絶対値を返す。

戻 り 値——*n* の int 型絶対値を返す。

注　　意——通常，abs 関数はマクロとして定義されるが，__NO_STDLIB_INLINE__が定義された場合には実体をもつ関数となる。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——div，labs，ldiv，wabs

access

用　途——ファイルにアクセスできるかどうかを調べる。

書　式——#include <unistd.h>

```
int access (const char *path, int amode);
```

解　説——access 関数は *path* で指定されたファイルに，*amode* で指定されたアクセスができるかどうかを調べる。*amode* には次の 4 種類があり，それぞれの値の論理和を指定する。

- R_OK　読み込みの可否
- W_OK　書き込みの可否
- X_OK　実行の可否
- F_OK　ファイルの有無

戻り値——指定したアクセスが可能ならば 0 を，不可能ならば−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ファイルが存在しない
- EINVAL　*amode* に不正な値を指定した
- ELOOP　シンボリックリンクのネストが深すぎるか，ループしている

互換性——**Human68k** にはユーザ ID やグループ ID の概念がないので，access 関数がこれらの値に影響されることはない。

Human68k のファイル管理方法では読み込み禁止ファイルを作成することはできない。よってすべてのファイルは読み込み可能であると仮定する。またファイルが実行可能であるかどうかは，ファイル名の拡張子が“.x”，“.r”，“.z”，“.bat”であるか，実行属性ビットが設定されているかどうかで判定する。

ただし，ディレクトリが実行可能とはディレクトリ内部へ移動することができるということであり，**Human68k** ではつねに可能であると仮定する。

規　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——chmod，chown，stat

変　更——従来では拡張子“.Z”をもつものも実行ファイルとして認識していたが，現在の *LIBC* では認識しない。“.Z”という拡張子は確かに **Human68k** での実行ファイルにも使われるがその頻度は少なく，むしろ compress プログラムによる圧縮データを表すことのほうが多いためである。

acos

用　途――逆余弦 $arccos(x)$ を求める。

書　式――
```
#include <math.h>
double acos (double x);
#include <sys/xmath.h>
double _f_acos (double x);
double _fe_acos (double x);
double _fpu_acos (double x);
```

解　説――acos 関数は x の逆余弦(アークコサイン)を計算する。各関数はそれぞれ次のように動作する。

- acos　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_acos　数値演算コプロセッサ命令を直接ドライブする
- _fe_acos　FLOAT パッケージを呼び出す
- _fpu_acos　I/O 数値演算コプロセッサを直接ドライブする

戻り値――正しく値が求められた場合，acos 関数は $-1 \leq x \leq 1$ の範囲内で x の逆余弦を返す。結果は，$0 \leq acos(x) \leq \pi$（単位：ラジアン）の範囲の値である。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して，同じく非数を返す。

- EDOM　x の値が非数，または x の値が計算範囲外

注　意――<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは acos は _fe_acos の別名となり，__DIRECT_IOFPU__が定義されているときは acos は _fpu_acos の別名となる。また，__DIRECT_FPU__が定義されているときは acos は _f_acos の別名となる。

規　格――***Project LIBC Group, ANSI C, POSIX.1, XPG3, AES/OS, 4.3BSD, SYSV***

関連項目――cos, isnan

acosh

用　途——双曲逆余弦 $arccosh(x)$ を求める。

書　式——

```
#include <math.h>
double acosh (double x);
#include <sys/xmath.h>
double _f_acosh (double x);
double _fe_acosh (double x);
double _fpu_acosh (double x);
```

解　説——acosh 関数は x の双曲逆余弦 (ハイパボリックアークコサイン) を計算する。各関数はそれぞれ次のように動作する。

- acosh　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_acosh　数値演算コプロセッサ命令を直接ドライブする
- _fe_acosh　FLOAT パッケージを呼び出す
- _fpu_acosh　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，acosh 関数は $x \geq 1$ の範囲で x の双曲逆余弦を返す。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　x の値が非数，または x の値が計算範囲外

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは acosh は _fe_acosh の別名となり，__DIRECT_IOFPU__が定義されているときは acosh は _fpu_acosh の別名となる。また，__DIRECT_FPU__が定義されているときは acosh は _f_acosh の別名となる。

規　格——*Project LIBC Group*

関連項目——cosh，isnan

alarm

用　途――アラームシグナルを設定する。

書　式――#include <unistd.h>

```
unsigned int alarm (unsigned int seconds);
```

解　説――alarm関数は，*seconds*秒後に現在実行中のプロセスに対してアラームシグナル(SIGALRM)が発行されるようにリアルタイムクロックを設定する。ただし*seconds*が0の場合は，現在設定されているアラームシグナルの予約を解除する。

アラームシグナルは多重に設定することはできず，つねに一番最後に設定されたものが有効となる。

またspawnやexec系の関数，system関数などで子プロセスが起動されると，その時点でアラームシグナルは中断され，子プロセス終了後に再度カウントを再開する。したがって，このような場合には正確な時間の計測は行えない。

戻り値――現在アラームシグナルが予約されている場合は，そのシグナルが何秒後に生成される予定なのかを数値で返し，予約されていないか，すでに発行後ならば0を返す。

注　意――アラームシグナルは，リアルタイムクロック(RTC)割り込みを用いて実現されている。したがってalarm関数を用いてアラームシグナルを予約すると，実際にシグナルが発行されるまで，$1/16$秒ごとに1回の割り込みが起こる。だが，割り込み処理は非常に軽いので，パフォーマンスには実質的な影響はない。しかし，それだけの回数で割り込みが発生しているということについては注意する必要がある。

また，割り込みがDOSコールの処理中に発生した場合は，安全のために$1/16$秒シグナルを遅らせる。

規　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――exec，pause，raise，sigaction，spawn，system

alloca

用　途——スタックフレームからメモリを割り当てる。

書　式——#include <alloca.h>

```
void *alloca (size_t size);
```

解　説——alloca 関数は少なくとも *size* バイトの**メモリブロック**を，実行中の関数のスタックフレームから割り当て，そのメモリブロックへのポインタを返す。メモリブロックは適切にアラインメント調整されるので，どんなポインタにでもキャストすることができる。

なお割り当てられたメモリブロックは関数のスタックフレームに位置し，alloca 関数を呼び出した関数がより上位の関数(呼び出し元)に return すると自動的に解放される。

戻り値——正常にメモリブロックを割り当てられた場合は，そのメモリブロックへのポインタを返し，失敗した場合は NULL を返す。

注　意——alloca 関数を呼び出した関数が上位の関数(呼び出し元)へ return した後は，その関数内で割り当てたメモリブロックへのアクセスは保証されない。なぜならば，スタックフレームは関数の呼び出しごとに破壊されるからである。

alloca 関数は **GCC のビルトイン関数**を用いて実装されており，実体のある関数ではない。また**スタックサイズ**固定なので，alloca 関数を頻繁に使用するプログラムに対してはあらかじめ十分なサイズのスタック領域を与えておく必要がある。万一，スタックオーバーフローが起こった場合，その動作は不定であり，一切保証されない。

スタックサイズは，プログラムを起動する際にコマンドラインから指定することもできる(_enargv 関数参照)が，あらかじめサイズを指定したい場合は，次のように変数_stacksize をユーザが明示的に定義しなければいけない。

```
int _stacksize = 128 * 1024; /* 128Kbyte */
```

通常，alloca 関数はスタックチェックを行わず，NULL を返すことはない。しかし<alloca.h>が読み込まれる前に__ALLOCA_STACK_CHECK__が定義された場合，alloca 関数は呼び出されるたびにスタックチェックを行うようになる。少なくとも alloca 関数に限っては，**スタックオーバーフロー**を起こさないようになる。

規　格——***Project LIBC Group***

関連項目——_enargv，free，malloc，realloc

asctime

用　　途――日付データを文字列に変換する。

書　　式――#include <time.h>

```
char *asctime (const struct tm *timeptr);
```

解　　説――asctime関数は*timeptr*が指すtm構造体の内容(UTC)を，次のような形式の文字列に変換する。

```
Tue Nov 19 21:03:20 1991\n\0
```

戻り値――変換が成功した場合は変換された文字列へのポインタを返し，失敗した場合はNULLを返す。

注　　意――結果はgmtime関数内部の静的領域に格納されるため，localtime関数，gmtime関数，asctime関数，ctime関数のいずれかの関数を呼び出すたびに内容が上書きされることに注意すること。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――ctime，gmtime，localtime

asin

用　途——逆正弦 $arcsin(x)$ を求める。

書　式——

```
#include <math.h>
double asin (double x);
#include <sys/xmath.h>
double _f_asin (double x);
double _fe_asin (double x);
double _fpu_asin (double x);
```

解　説——asin 関数は x の逆正弦 (アークサイン) を計算する。各関数はそれぞれ次のように動作する。

- asin　　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_asin　　数値演算コプロセッサ命令を直接ドライブする
- _fe_asin　　FLOAT パッケージを呼び出す
- _fpu_asin　　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，asin 関数は $-1 \leq x \leq 1$ の範囲で x の逆余弦を返す。結果は，$-\pi/2 \leq asin(x) \leq \pi/2$ (単位：ラジアン) の範囲の値である。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　x の値が非数，または x の値が計算範囲外

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは asin は _fe_asin の別名となり，__DIRECT_IOFPU__が定義されているときは asin は _fpu_asin の別名となる。また，__DIRECT_FPU__が定義されているときは asin は _f_asin の別名となる。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——isnan，sin

asinh

用　途——双曲正弦 $arcsinh(x)$ を求める。

書　式——

```
#include <math.h>
double asinh (double x);
#include <sys/xmath.h>
double _f_asinh (double x);
double _fe_asinh (double x);
double _fpu_asinh (double x);
```

解　説——asinh 関数は x の双曲逆正弦 (ハイパボリックアークサイン) を計算する。各関数はそれぞれ次のように動作する。

- asinh　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_asinh　数値演算コプロセッサ命令を直接ドライブする
- _fe_asinh　FLOAT パッケージを呼び出す
- _fpu_asinh　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，asinh 関数は x の双曲逆正弦を返す。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　x の値が非数だった

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは asinh は _fe_asinh の別名となり，__DIRECT_IOFPU__が定義されているときは asinh は _fpu_asinh の別名となる。また，__DIRECT_FPU__が定義されているときは asinh は _f_asinh の別名となる。

規　格——*Project LIBC Group*

関連項目——isnan，sinh

assert

用　途——プログラム診断を行う。

書　式——

```
#include <assert.h>
void assert (int expression);
```

解　説——assert 関数はプログラム中にプログラム診断文を挿入するときに用いる。つまり，assert 関数は *expression* で与えられた式を評価し，偽 (0) だった場合には標準エラーメッセージに表示した後，abort 関数を呼び出してプロセスを異常終了させる。

ただし，コンパイル時において<assert.h>を読み込む前に NDEBUG が定義されていると，assert 関数はただの空文と置き換えられ，診断文は取り除かれる。

たとえば "foo.c" というプログラムの 50 行目に，assert 関数を次のように使用するとしよう。

```
assert ((x + y) < 3);
```

これは，50 行目の時点で x と y の和が 3 より小さくなくてはならないことを意味する。にもかかわらず，何らかの問題によって式 (x + y) < 3 の評価結果が偽になった場合，assert 関数は次のようなメッセージを出力して，プロセスを異常終了させる。

```
Assertion failed: <(x + y) < 3> at line 50 in file foo.c
```

戻り値——なし。式の評価結果が偽ならば，assert 関数は決して戻ってこない。

注　意——NDEBUG は，<assert.h>を読み込む前に定義されていなくてはならない。

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——abort

atan

用　途——逆正接 $arctan(x)$ を求める。

書　式——

```
#include <math.h>
double atan (double x);
#include <sys/xmath.h>
double _f_atan (double x);
double _fe_atan (double x);
double _fpu_atan (double x);
```

解　説——atan 関数は x の逆正接(アークタンジェント)を計算する。各関数はそれぞれ次のように動作する。

- atan　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_atan　数値演算コプロセッサ命令を直接ドライブする
- _fe_atan　FLOAT パッケージを呼び出す
- _fpu_atan　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，atan 関数は $-1 \leq x \leq 1$ の範囲で x の逆正弦を返す。結果は $-\pi/2 \leq atan(x) \leq \pi/2$ (単位：ラジアン) の範囲の値である。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　x の値が非数，または x の値が計算範囲外

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは atan は_fe_atan の別名となり，__DIRECT_IOFPU__が定義されているときは atan は_fpu_atan の別名となる。また，__DIRECT_FPU__が定義されているときは atan は_f_atan の別名となる。

規　格——*Project LIBC Group*, *ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——isnan, tan

atan2

用　途——逆正接 $arctan(y/x)$ を求める。

書　式——

```
#include <math.h>
double atan2 (double y, double x);
#include <sys/xmath.h>
double _f_atan2 (double y, double x);
double _fe_atan2 (double y, double x);
double _fpu_atan2 (double y, double x);
```

解　説——atan2 関数は y/x の逆正接 (アークタンジェント) を計算する。各関数はそれぞれ次のように動作する。

- atan2　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_atan2　数値演算コプロセッサ命令を直接ドライブする
- _fe_atan2　FLOAT パッケージを呼び出す
- _fpu_atan2　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，atan2 関数は x の逆正接を返す。戻り値の象限は，y，x の符号によって決定する。結果は $-\pi \leq atan2(y,x) \leq \pi$ (単位：ラジアン) の範囲の値である。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　y，x の値が非数，x の値が 0 または y，x の値が計算範囲外

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは atan2 は _fe_atan2 の別名となり，__DIRECT_IOFPU__が定義されているときは atan2 は _fpu_atan2 の別名となる。また，__DIRECT_FPU__が定義されているときは atan2 は _f_atan2 の別名となる。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——atan，isnan，tan

atanh

用　　途——双曲逆正接 $arctanh(x)$ を求める。

書　　式——

```
#include <math.h>
double atanh (double x);
#include <sys/xmath.h>
double _f_atanh (double x);
double _fe_atanh (double x);
double _fpu_atanh (double x);
```

解　　説——atanh 関数は x の双曲逆正接(ハイパボリックアークタンジェント)を計算する。各関数はそれぞれ次のように動作する。

- atanh　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_atanh　数値演算コプロセッサ命令を直接ドライブする
- _fe_atanh　FLOAT パッケージを呼び出す
- _fpu_atanh　I/O 数値演算コプロセッサを直接ドライブする

戻 り 値——正しく値が求められた場合，atanh 関数は $-1 < x < 1$ の範囲で x の双曲逆正接を返す。もし x が非数 (NaN) だったならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　x の値が非数，または x の値が計算範囲外

注　　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは atanh は _fe_atanh の別名となり，__DIRECT_IOFPU__が定義されているときは atanh は _fpu_atanh の別名となる。また，__DIRECT_FPU__が定義されているときは atanh は _f_atanh の別名となる。

規　　格——*Project LIBC Group*

関連項目——isnan，tanh

atexit

用　　途――プロセス終了時に呼び出される関数を登録する。

書　　式――

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

解　　説――atexit 関数は，*func* で指定した関数をプロセスの正常終了時に呼び出される関数のリストに登録する。プロセスが正常終了すると，これら登録された関数は，登録された順番とは逆の順番(つまり最後に登録した順番)に引数なしで呼び出される。

正常終了とは，exit 関数によってプロセスを終了するか，main 関数から return した場合のことであり，abort 関数で終了させたり何らかのエラーで強制終了させられた場合のことではない。なお登録できる関数の数は，最大 ATEXIT_MAX 個までである。ATEXIT_MAX は<limits.h>に定義されている。

戻 り 値――正常に登録できた場合は 0 を返し，失敗した場合には 0 以外の値を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　　すでに ATEXIT_MAX 個の関数が登録されているので，これ以上登録することはできない

規　　格――*ANSI C*，*SYSV*

関連項目――abort，exit，onexit，sysconf

atof

用　　途――文字列を double 型倍精度浮動小数に変換する。

書　　式――#include <stdlib.h>

```
double atof (const char *nptr);
```

解　　説――atof 関数は，*nptr* で指定された文字列を double 型倍精度浮動小数に変換する。変換は先頭の空白は無視し，null 文字に出会うか，数値に変換できない文字に出会うまで実行される。数値の最後には指数があってもよく，文字“e”か“E”に続く符合 (省略可能) と指数値を正しく認識する。

```
-3.1415926, 1.8794E+15, 2.42e-6
```

小数点として認識する文字は，ロケールの LC_NUMERIC カテゴリに影響される。

戻 り 値――変換した結果を返す。

注　　意――変換した結果が double 型倍精度浮動小数で表現しきれない場合，その結果は不定となる。*LIBC* は C ロケールしかサポートしていない。

規　　格――***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――atoi，atol，strtod，strtol，strtoul

atoi

用　　途——文字列を符合つき int 型整数に変換する。

書　　式——#include <stdlib.h>

```
int atoi (const char *nptr);
```

解　　説——atoi 関数は *nptr* で指定された文字列を符合つき int 型整数に変換する。変換は先頭の空白は無視し，null 文字に出会うか，数値に変換できない文字に出会うまで行われる。

戻 り 値——変換した結果を返す。

注　　意——変換した結果が符合つき int 型整数で表現しきれない場合，その結果は不定となる。

規　　格——*ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——atof, atol, strtod, strtol, strtoul

atol

用　　途――文字列を符合つきlong型整数に変換する。

書　　式――#include <stdlib.h>

```
long atol (const char *nptr);
```

解　　説――atol関数は，*nptr*で指定された文字列を符合つきlong型整数に変換する。変換は先頭の空白は無視し，null文字に出会うか，数値に変換できない文字に出会うまで行われる。

戻 り 値――変換した結果を返す。

注　　意――変換した結果が符合つきlong型整数で表現しきれない場合，その結果は不定となる。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――atof，atoi，strtod，strtol，strtoul

B

bcmp

用　途——2つの領域の内容を比較する。

書　式——#include <string.h>

```
int bcmp (const void *region2, const void *region1, size_t n);
```

解　説——bcmp 関数は，*region1* と *region2* で指定された領域の内容を *n* バイト分比較する。bcmp 関数は strcmp 関数とは異なり，null 文字を検出しても処理を中断しない。

戻り値——比較の結果，2つの文字列がまったく同じならば bcmp 関数は 0 を返す。異なる場合，その位置での *region1* 側の文字が *region2* 側の文字よりも大きい場合は正の値を，小さい場合は負の値を返す。

注　意——1文字は1バイトとなる。

規　格——*4.3BSD*

関連項目——memcmp，strcmp，strcoll，strncmp，strxfrm

bcopy

用　　途――領域をコピーする。

書　　式――#include <string.h>

```
void *bcopy (void *region2, const void *region1, size_t n);
```

解　　説――bcopy関数は，*region2*の指す領域から*n*バイトを*region1*の指す領域にコピーする。bcopy関数はstrcpy関数とは異なり，null文字を検出しても処理を中断しない。

戻 り 値――*region2*へのポインタを返す。

注　　意――領域が重なっていた場合の動作は未定義である。*region1*は，*n*バイトのデータを格納するのに十分な領域を指していなければならない。1文字は1バイトとなる。

規　　格――***4.3BSD***

関連項目――memcpy，memmove，strcpy，strncpy

brk

用　途——ブレーク値を設定する。

書　式——
```
#include <stdlib.h>
int brk (void *addr);
```

解　説——brk関数は現在のブレーク値を*addr*で指定した値に設定する。ただし*addr*は，*addr*以上のページ境界（*LIBC*では疑似的に4096バイトをページサイズとする）に整列される。

ブレーク値とはプログラムのヒープ領域の最後尾のことをいい，ヒープ領域が負になったり，setrlimit関数で指定されたリミット値を越えない範囲で変更することができる。またこのブレーク値は，malloc関数などで新しいメモリを確保していくことで自動的に拡張される。

戻り値——正常にブレーク値を設定できた場合は0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなったか，制限値に達した

注　意——*LIBC*では，プログラムのもつヒープ領域はプログラムの最後から連続して確保されていなくてはならない。したがって常駐プロセスなどがプロセスの後ろなどにロードされると，現在のメモリブロックの自由な拡張が阻害され，その結果，十分なフリーエリアがあるにもかかわらずメモリが足りなくなることがある。

brk関数はブレーク値を変更することでヒープ領域のサイズを変更することができるが，ヒープ領域がどのように使用されているかについては関知しない。したがって，現在のサイズよりも縮小した場合には，malloc関数などで得たメモリ領域が破壊されることがあるので注意すること。

規　格——***Project LIBC Group*，*XC*，*4.3BSD***

関連項目——chkml，rbrk，sbrk，sizmem

bsearch

用　途——ソート済みの配列に対してバイナリサーチを行う。

書　式——
```
#include <stdlib.h>
void *bsearch (const void *key, const void *base,
               size_t nmemb, size_t size,
               int (*compare) (const void *, const void *));
```

解　説——bsearch 関数は，*base* が指す任意の配列から *key* で指定されたデータをバイナリサーチによって検索し，それが見つかった位置へのポインタを返す。検索される配列は，*base* から始まる *nmemb* 個の項目（1項目は *size* バイト）であり，すでに比較関数 *compare* の仕様にしたがって正順にソートされていなければならない。

bsearch 関数は検索を行うにあたり，必要に応じて関数 *compare* を用い，任意の2項目についてその大小を比較する。*compare* で指定した関数は，与えられた2項目を任意のアルゴリズムで比較し，1つ目の項目のほうが大きければ正の値，2つ目の項目のほうが大きければ負の値を，等しければ0を返す必要がある。

戻り値——求めるデータが見つかった場合はその配列中の位置へのポインタを返し，見つからなかった場合には NULL を返す。

注　意——配列はすでに正順にソートされていなければならない。もし正しくソートされていない場合は求めるデータが見つからないなど，どのような結果になるかについては未定義である。

配列中に求めるデータが2つ以上ある場合，bsearch 関数がどちらのデータへのポインタを返すかは未定義であり，状況によって変化する。

規　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

サンプル——List 1-1 ● 比較関数の例

```
/*
** 任意の２つの整数を比較するための比較関数の例。
** compare_int (const void *data1, const void *data2);
** １番目の項目が大きければ正，逆なら負，等しければ０を返す。
*/

int compare_int (const void *data1, const void *data2)
{
    const int *i1 = (const int *) data1;
    const int *i2 = (const int *) data2;
    return (int) (*i1 - *i2);
}
```

bzero

用　　途——領域を 0 で埋める。

書　　式——#include <string.h>

```
void *bzero (void *region, size_t n);
```

解　　説——bzero 関数は *region* の指す領域から *n* バイトを 0 で埋める。bzero 関数は strset 関数とは異なり，null 文字を検出しても処理を中断しない。

戻 り 値——*region* へのポインタを返す。

注　　意——1 文字は 1 バイトとなる。

規　　格——*4.3BSD*

関連項目——memchr, memcmp, memcpy, memmove, memset

calloc

用　　途――メモリブロックを確保する。

書　　式――#include <stdlib.h>

```
void *calloc (size_t nmember, size_t size);
```

解　　説――calloc 関数は，1項目が *size* バイトのデータ *nmember* 個分の配列に相当するメモリ領域を**ヒープ領域**から確保し，そのメモリ領域へのポインタを返す。確保されたメモリ領域は，あらかじめすべて0でクリアされる。

また確保されたメモリ領域の先頭アドレスは，いかなるデータ型にでもキャストできる可能なように CPU 依存の**アラインメント**に調整される。

戻 り 値――正常に確保できた場合はその領域へのポインタを返し，失敗した場合には NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなったか，制限値に達した

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――brk，free，malloc，rbrk，realloc，sbrk

ceil

用　途——x 以上の整数のなかで最も小さな数を返す。

書　式——#include <math.h>

```
double ceil (double x);
```

解　説——ceil 関数は x 以上の値をもつ整数のなかで最も小さな数を返す。

戻り値——正しく値が求められた場合はその値を返す。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　x の値が非数，または±HUGE_VAL である

注　意——つねに FLOAT パッケージを呼び出す。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——fabs，floor，fmod

cgets

用　　途——コンソールから直接1行入力する。

書　　式——#include <conio.h>

```
char *cgets (char *buff);
```

解　　説——cgets関数はコンソールから1行分のデータを入力し，*buff*が指す領域に格納する。ただし*buff* [0] には，読み込みを行う領域のサイズをunsigned char型のデータで設定しておく必要がある。

読み込みは指定したバイト数を読み込むか改行コードに出会うまで行われ，実際に読み込まれたバイト数が*buff* [1] に設定される。したがって読み込まれたデータは，*buff* [2] から格納されることになる。

cgets関数はつねにコンソールから直接データを読み取るため，標準入力の状態に左右されることも，stdioライブラリによるバッファリングの作用も受けない。また，cgets関数は1行を入力するまで待つが，すでにキーボードバッファにデータがある場合やungetch関数によって押し戻されたデータがある場合は，それらからもデータを取得する。

戻り値——*buff*[2] のアドレスを返す。エラーはない。

注　　意——読み込みバイト数は，unsigned char型で表現できる範囲に限られる。また，構造上0x00の文字コードは取得することができない。

*buff*は結果を格納するだけの十分な大きさの領域を指していなければならない。

規　　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——fgets，getch，getche，gets

chdir

用　途——カレントワーキングディレクトリを変更する。

書　式——
```
#include <unistd.h>
int chdir (const char *path);
```

解　説——chdir 関数は，カレントワーキングディレクトリを *path* で指定したディレクトリに変更する。

戻り値——正しくワーキングディレクトリを変更できた場合は 0 を，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　*path* で指定したディレクトリが存在しない
- ENOTDIR　*path* で指定したパス名にディレクトリ以外の要素が含まれている
- ELOOP　シンボリックリンクのネストが深すぎるか，ループしている

互換性——*path* にドライブ名が含まれていた場合，*XC*，*MS–C 7.0* などではそのドライブのカレントワーキングディレクトリを変更するだけだが，*LIBC* では同時にカレントドライブをそのドライブに変更する。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——chdrive，getcwd，getdcwd

chdrive

用　途――カレントドライブを変更する。

書　式――#include <unistd.h>

```
int chdrive (int drive);
```

解　説――chdrive関数は，カレントドライブを*drive*で指定したドライブに変更する。*drive*はA:ドライブを1とし，以降2がB:ドライブ，3がC:ドライブという順番に指定する。

戻り値――正しくドライブを変更できた場合は0を返し，失敗した場合には-1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EXDEV　不正なドライブ*drive*を指定した

規　格――***Project LIBC Group，　MS-C 7.0***

関連項目――chdir，getcwd，getdcwd

C

chkml

用　　途——空きメモリ容量をバイト単位で調べる。

書　　式——#include <stdlib.h>

```
size_t chkml (void);
```

解　　説——chkml 関数は空きメモリ容量，正確には連続して確保できる最大の空きメモリブロックのサイズを調べ，結果をバイト単位で返す。

戻 り 値——空きメモリ容量をバイト単位で返す。

注　　意——正確な空きメモリ容量とは異なる。また，ここでいう空きメモリブロックとは **Human68k** が管理しているメモリブロックのことであり，プロセスのヒープ領域から確保されるメモリブロック (管理するのはプロセス自身) ではない。

規　　格——*Project LIBC Group*，*XC*

関連項目——brk，sizmem

chmod

用　　途——ファイルのアクセスモードを変更する。

書　　式——#include <stat.h>

```
int chmod (const char *name, mode_t mode);
```

解　　説——chmod関数は*name*で指定したファイルのファイルモードを，*mode*で指定する値に変更する。*mode*に指定することができるのは次のとおりで，それぞれの値の論理和を指定する。

- S_IEXEC　実行ファイル，チェンジディレクトリ可能
- S_IWRITE　書き込み可
- S_IREAD　読み込み可

戻 り 値——正常にアクセスモードの変更ができた場合は0を返し，失敗した場合は−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- ENOENT　ファイルが見つからない
- ENOENT　不正なドライブを指定した
- EPERM　ボリュームファイルを変更しようとした

互 換 性——**Human68k**ではファイルに実行属性や読み込み禁止属性を設定することはできないので(実行属性についてはexecdを使えば可能)，S_IEXECやS_IREADは疑似的に作り出される値である。

規　　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——fchmod，fstat，_mode2dos，stat

変　　更——従来ではS_IEXECによってファイルの実行属性を設定していたが(実行属性については execd の機能)，これは標準的な環境では余計な動作となるため廃止した。代わりに現在の***LIBC***では，S_IEXBIT を指定することで実行属性だけを個別に設定できるようにした。

- S_IEXBIT　実行属性ビット

chown

用　途——ファイルのオーナーおよびグループを変更する。

書　式——#include <unistd.h>

```
int chown (const char *path, uid_t owner, gid_t group);
```

解　説——chown 関数は，*path* で指定されたファイルのオーナーIDおよびグループIDを変更する。

戻り値——正常に変更できた場合は0を返し，失敗した場合には−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ファイルが存在しない
- ELOOP　シンボリックリンクのネストが深すぎるか，ループしている

互換性——**Human68k** にはユーザIDやグループIDの概念がないので，chown 関数がこれらの値に影響されることはない。また *LIBC* 内では，デフォルトの仮想ユーザIDおよび仮想グループIDを root と仮定しているため，chown 関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　格——***Project LIBC Group***, ***POSIX.1***, ***XPG3***, ***AES/OS***, ***4.3BSD***, ***SYSV***

関連項目——chmod, fchmod, fchown

chsize

用　途――ファイルの長さを変更する。

書　式――#include <unistd.h>

```
#include <unistd.h>
int chsize (int fildes, int len);
```

解　説――chsize 関数は，*fildes* が指すファイルの長さを *len* バイトに変更する。もし *len* が現在のファイルサイズよりも短い場合は，その間のデータはすべて捨てられ，反対に長い場合はすべて0のデータで埋められる。

ただし，chsize 関数はキャラクタデバイスに対しては動作しない。また，ファイルは書き込み可能であること。

戻り値――正常に変更できた場合は0を返し，失敗した場合には−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- EROFS　リードオンリーファイルシステムである

規　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――creat，ftruncate，open，truncate

clearenv

用　途——プロセスの環境変数テーブルをクリアする。

書　式——

```
#include <stdlib.h>
int clearenv (void);
```

解　説——clearenv 関数はプロセスの**環境変数テーブル**をクリアする。つまり，clearenv 関数を実行した後はすべての環境変数が未定義となる。

戻り値——正しくテーブルをクリアできた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　意——クリアされる環境変数テーブルはプロセス内部にコピーされた環境変数テーブルであり，親プロセスの環境変数テーブルや **Human68k** の環境変数テーブルはクリアされない。

clearenv 関数を実行すると変数 environ が変化するため，以前のポインタ値に対するアクセスは保証されない。また，*LIBC*のなかには環境変数を参照する関数もあるが，参照がどの時点で行われるかは関数に依存するので，必ずしもクリアすることによって影響を受けるとは限らない。

規　格——***Project LIBC Group***，***POSIX.1***

関連項目——environ，exec，getenv，putenv，spawn

clearerr

用　　途――ファイルストリームのエラーおよび終端指示子をクリアする。

書　　式――#include <stdio.h>
```
#include <stdio.h>
void clearerr (FILE *stream);
```

解　　説――clearerr関数は，*stream*が指すファイルストリームのエラー指示子および終端指示子をクリアする。ファイルストリーム関数のなかには，これらの指示子の状態によってはつねにエラーとなるものがある。

戻 り 値――clearerr関数には戻り値はない。

注　　意――ファイルストリームにエラー指示子が設定されている場合，変数errnoにその原因を示すエラーコードを設定する。ただし変数errnoの値は，ファイル操作の直後しか保証されない。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――feof，ferror

clock

用　途——起動してから現在までの経過時間を測定する。

書　式——#include <time.h>

```
clock_t clock (void);
```

解　説——clock 関数は，プログラムが起動してから呼び出された時点までの経過時間を返す。clock 関数で使用する値は，マシンが起動してからの経過時間を 10 ミリ秒単位で表したものである。秒単位の実時間を求めるには，この値を CLOCKS_PER_SEC で割ればよい。

戻り値——プログラムが起動してから clock 関数が呼び出されるまでの経過時間を，10 ミリ秒単位で返す。

注　意——プログラムが起動してから 24 時間以上経過すると，正しい経過時間を得ることができない。古い *ANSI C* 規格では，CLOCKS_PER_SEC は CLK_TCK となっていた。

規　格——*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

close

用　　途――ファイルをクローズする。

書　　式――

```
#include <unistd.h>
int close (int fildes);
```

解　　説――close 関数は，*fildes* で指定したファイルハンドルが指すファイルをクローズする。その結果，*fildes* を通してのファイルアクセスはできなくなる。また，クローズする前に unlink 関数によって削除を予約されたファイルは，close 関数が実行されると実際に削除される。

戻 り 値――正常にクローズできた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した

注　　意――close 関数と unlink 関数は **UNIX** 上の unlink システムコールをシミュレートしているが，**Human68k** のファイルシステムではリンクカウントを保持できないため，つねにカウントは 1 として扱っている。

また，その実現方法は完全ではないので，理論的には削除予約したはずのファイルが正しく削除されずに残ってしまうこともありえる。したがって unlink 関数は，ファイルをクローズした後に行うのが望ましいだろう。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――creat，dup，fclose，fcntl，fopen，open，unlink

closedir

用　途——ディレクトリストリームをクローズする。

書　式——#include <dirent.h>

```
void closedir (DIR *dirp);
```

解　説——closedir 関数は，*dirp*で指定するディレクトリストリームをクローズし，構造体を解放する。

戻り値——なし。

規　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——closedir，readdir，rewinddir，seekdir，telldir

commit

用　途――ファイルアクセスのバッファをフラッシュする。

書　式――
```
#include <unistd.h>
int commit (int fildes);
```

解　説――commit 関数は，*fildes* で指定したファイルハンドルが指すファイルをアクセスするために用いられているファイルバッファをフラッシュし，内容をディスク上に反映させる。

戻り値――正常にフラッシュできた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した

注　意――commit 関数が影響をおよぼすファイルバッファとは，“config.sys” の FILES= 行で指定される **Human68k** の内部バッファであり，*LIBC* がもつバッファでもなければ stdio ライブラリのそれでもない。

互換性――**Human68k** では，このバッファフラッシュはファイルハンドルごとに個別に行うことができない。したがって commit 関数は，現在オープンされているすべてのファイルに対して影響をおよぼす。

規　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――creat，open，read，write

cos

用　　途——余弦 $cos(x)$ を求める。

書　　式——

```
#include <math.h>
double cos (double x);
#include <sys/xmath.h>
double _f_cos (double x);
double _fe_cos (double x);
double _fpu_cos (double x);
```

解　　説——cos 関数は x (単位：ラジアン) の余弦 (コサイン) を計算する。各関数はそれぞれ次のように動作する。

- cos　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_cos　数値演算コプロセッサ命令を直接ドライブする
- _fe_cos　FLOAT パッケージを呼び出す
- _fpu_cos　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，cos 関数は x の余弦を返す。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返し，それ以外では計算結果を返す。ただし計算結果は不正確な値になる。

- EDOM　x の値が非数，または±HUGE_VAL
- ERANGE　x の値が大きく計算結果の有効桁が一部失われる場合，または x の値が非常に大きくて計算結果の有効桁が完全に失われる場合

注　　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは cos は _fe_cos の別名となり，__DIRECT_IOFPU__が定義されているときは cos は _fpu_cos の別名となる。また，__DIRECT_FPU__が定義されているときは cos は _f_cos の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——acos，isnan

cosh

用　　途――双曲余弦 $cosh(x)$ を求める。

書　　式――

```
#include <math.h>
double cosh (double x);
#include <sys/xmath.h>
double _f_cosh (double x);
double _fe_cosh (double x);
double _fpu_cosh (double x);
```

解　　説――cosh 関数は x の双曲余弦（ハイパボリックコサイン）を計算する。各関数はそれぞれ次のように動作する。

- cosh　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_cosh　数値演算コプロセッサ命令を直接ドライブする
- _fe_cosh　FLOAT パッケージを呼び出す
- _fpu_cosh　I/O 数値演算コプロセッサを直接ドライブする

戻り値――正しく値が求められた場合，cosh 関数は x の双曲余弦を返す。もし x が非数（NaN）ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返し，それ以外では無限大を返す。

- EDOM　x の値が非数だった
- ERANGE　計算結果がオーバーフローした

注　　意――<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは cosh は _fe_cosh の別名となり，__DIRECT_IOFPU__が定義されているときは cosh は _fpu_cosh の別名となる。また，__DIRECT_FPU__が定義されているときは cosh は _f_cosh の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　　格――***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――acos，cos，isnan

cprintf

用　途――直接コンソールへフォーマット出力する。

書　式――#include <conio.h>

```
int cprintf (const char *format, ...);
```

解　説――cprintf 関数は，*format* で指定した表示フォーマットにしたがって引数をコンソールに直接表示する。cprintf 関数は，機能的には printf 関数と似ているが，stdio ライブラリによるバッファリングの作用は一切受けない。

また出力はつねにコンソールに対して行われるので，標準出力，標準エラー出力などの状態に左右されることはない。さらにテキスト，バイナリモードの区別もないので，行末の CRLF を LF に変換する作業も行われない。

表示フォーマットの指定は printf 関数を参照のこと。

戻り値――実際にコンソールに出力した文字数を返す。エラーはない。

規　格――*Project LIBC Group,　MS-C 7.0*

関連項目――fprintf，printf，sprintf

cputs

用　途——直接コンソールへ1行出力する。

書　式——#include <conio.h>

```
int cputs (const char *string);
```

解　説——cputs 関数は，*string* で指定した文字列をコンソールに直接表示する。cputs 関数は，機能的には puts 関数と似ているが，stdio ライブラリによるバッファリングの作用は一切受けない。

出力は null 文字に到達するまで行われる。また，つねにコンソールに対して行われるので，標準出力，標準エラー出力などの状態に左右されることはない。さらにテキスト，バイナリモードの区別もないので，行末の CRLF を LF に変換する作業も行われない。

cputs 関数は行末の CRLF を自動的に出力することはない。

戻り値——正常に出力できた場合は0を返す。エラーはない。

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——cprintf，fputs，puts

creat

用　途——新しいファイルの作成と書き込みモードでオープンする。

書　式——#include <fcntl.h>

```
int creat (const char *path, mode_t mode);
```

解　説——creat 関数は，*path*で指定されたファイルを *mode*で指定した**ファイルアクセスモード**で新規に作成し，そのファイルを書き込みモードでオープンする。ただしすでに同名のファイルが存在した場合は，そのファイルを一度サイズ 0 に切り詰めて内容を破棄してからオープンする。その場合ファイルのファイルアクセスモードは，すでに存在していたファイルのファイルアクセスモードと同じになり，*mode*は無視される。

*mode*に指定することができるファイルアクセスモードは次のとおり。

- S_IREAD　　読み込み可能 (つねにこれは指定される)
- S_IWRITE　　書き込み可能

creat 関数は open 関数に対して次の指定をした場合と同じなので，詳細な動作については open 関数の説明を参照のこと。

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

戻り値——正常にファイルをオープンできた場合はそのファイルハンドルを，失敗した場合は−1 を返し，変数 errno にその原因を示すエラーコードを設定する。

- EISDIR　ディレクトリである
- EMFILE　これ以上ファイルをオープンできない
- ENOSPC　ディスクがいっぱいである
- ELOOP　シンボリックリンクのネストが深すぎるか，ループしている

互換性——**Human68k** では読み込み禁止属性のファイルを作成することはできないので，S_IREAD の指定/未指定に関わらず，すべてのファイルは読み込み可能となる。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——open

ctermid

用　　途――現在のコントロール端末の名称を取得する。

書　　式――#include <unistd.h>

```
char *ctermid (char *s);
```

解　　説――ctermid関数はカレントプロセスのコントロール端末の名称をパス名の形で取得し，*s*がNULLでなければ*s*の指す領域に，*s*がNULLならば関数内部の静的領域にその結果をコピーする。

戻 り 値――正常に取得することができた場合，その結果がコピーされた領域へのポインタを返し，失敗した場合には空文字列がコピーされる。

注　　意――*s*がNULLでなければ，*s*は最低L_ctermidバイトの領域を指していなければならない。また*s*がNULLならば，ctermid関数の静的領域は呼び出しごとに上書きされることに注意すること。なおL_ctermidは<stdio.h>で定義されている。

互 換 性――**Human68k**ではコントロール端末という概念がないため，ctermid関数では標準入力のデバイスを調べ，ブロックデバイスならば“con”を，キャラクタデバイスならばその名前を返す。

規　　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――ttyname

ctime

用　　途——日付データを文字列に変換する。

書　　式——#include <time.h>

```
char *ctime (const struct tm *timeptr);
```

解　　説——ctime 関数は，*timeptr* が指す tm 構造体の内容 (UTC) を，**地域時間情報**の設定にしたがって**地域時間**に変換し，次のような形式の文字列に変換する。

```
Tue Nov 19 21:03:20 1991\n\0
```

ctime 関数は，地域時間情報を得るために tzset 関数を呼び出す。

戻 り 値——変換が成功した場合は変換された文字列へのポインタを返し，失敗した場合は NULL を返す。

注　　意——結果は gmtime 関数内部の静的領域に格納されるため，localtime 関数，gmtime 関数，asctime 関数，ctime 関数のいずれかの関数を呼び出すたびに内容が上書きされることに注意すること。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——asctime，gmtime，localtime

cuserid

用　　途——ユーザのログイン名を取得する。

書　　式——#include <unistd.h>

```
char *cuserid (char *s);
```

解　　説——cuserid関数は現在のユーザのログイン名を取得し，*s*がNULLでなければ*s*の指す領域に，*s*がNULLならばcuserid関数内部の静的領域に，その結果をコピーする。

戻 り 値——正常に取得することができた場合は，その結果がコピーされた領域へのポインタを返す。失敗した場合には，*s*がNULLでなければ空文字がコピーされ，*s*がNULLならばNULLを返す。

注　　意——*s*がNULLでなければ，*s*は最低L_cuseridバイトの領域を指していなければならない。また*s*がNULLならば，cuserid関数の静的領域は呼び出しごとに上書きされることに注意すること。なおL_cuseridは<stdio.h>で定義されている。

互 換 性——**Human68k**はシングルユーザ用であるため，ユーザ管理に関する概念がない。そのためcuserid関数では，ksh，bash，fish，minshなどのシェルと同様，環境変数USERあるいはLOGNAMEからログイン名を取得することにしている。

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——getlogin，getpwnam，getpwuid

_dehupair

用　途——コマンドライン文字列を個々の引数に分解する。

書　式——
```
#include <sys/xstart.h>
int _dehupair (const char *com, char *buff);
```

解　説——_dehupair 関数は *com* で指定したコマンドライン文字列を解析し，1 つの引数が 1 つの文字列となるようにし，その結果を *buff* が指す領域に格納する。

各引数はスペースによって区切られ，引数の頭の余分な空白はすべて取り除かれる。ただし，ダブルクォーテーションマークやシングルクォーテーションマークによって必要な部分をクォートすることができる。

たとえば，*com* が次のような文字列を指していた場合，

```
arg1    arg2 'arg3 arg4' arg5\0
```

結果として *buff* の指す領域には，次のようなデータが格納されることになる。

```
arg1\0arg2\0arg3 arg4\0arg5\0
```

戻り値——文字列を分解した結果，取り出せた引数の数を返す。引数が 1 つもなかった場合は 0 を返す。

規　格——***Project LIBC Group***

関連項目——_enargv，_main，_start

_dellastsep

用　　途――パス名の最後のパス区切り記号を削除する。

書　　式――

```
#include <sys/xglob.h>
char *_dellastsep (char *path);
```

解　　説――_dellastsep 関数は，*path* で指定されたパス名の最後のパス区切り記号を削除する。ただし，区切り記号が存在しない場合は何も行わない。

戻 り 値――*path* を返す。

注　　意――*LIBC* ではパス区切り記号は "/" と "\" の両方を認識するが，結果はすべてスラッシュ "/" になるように変換される。

_dellastsep 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格――***Project LIBC Group***

関連項目――_addlastsep，_tobslash，_toslash

difftime

用　　途——2つの時刻の差を計算する。

書　　式——`#include <time.h>`

```
double difftime (time_t newtime, time_t oldtime);
```

解　　説——`difftime` 関数は *oldtime* から *newtime* にいたるまでの時間を秒単位で計算し，その結果 (差) を倍精度浮動小数に変換して返す。

戻 り 値——秒単位の時間差 *oldtime* − *newtime* を倍精度浮動小数に変換して返す。

注　　意——`difftime` 関数は，デフォルトではマクロとして定義され，実体をもたないが，`__NO_TIME_INLINE__` が定義された場合は実体をもつ関数となる。

規　　格——*ANSI C*， ***POSIX.1***， ***XPG3***， ***AES/OS***， ***4.3BSD***， ***SYSV***

関連項目——`clock`，`time`

div

用　　途——int 型整数の除算を行う。

書　　式——#include <stdlib.h>

```
div_t div (int number, int denom);
```

解　　説——div 関数は int 型の整数である *number* を *denom* で割った商と剰余を求める。商と剰余の符合は，引数の符合が同じであれば正，異なれば負となる。また *denom* で割り切れない場合，計算結果は代数学上の商よりも小さい最大の整数となる。

戻 り 値——除算の結果を div_t 構造体に代入し，その構造体を返す。構造体の定義は次のとおりで，quot に商を，rem に剰余を格納する。

```
struct {
    int quot;   /* 商 */
    int rem;    /* 剰余 */
};
```

注　　意——変換した結果が int 型で表現しきれない場合，その結果は不定である。

規　　格——*ANSI C*，　*SYSV*

関連項目——floor，ldiv

drand

用　途——実数の乱数を生成する。

書　式——`#include <stdlib.h>`

```
double drand (void);
```

解　説——drand 関数は $0.0 \leq x \leq 1.0$ の範囲で実数の乱数を生成し，その値を返す。

戻り値——$0.0 \leq x \leq 1.0$ の範囲の乱数を double 型で返す。

注　意——乱数シードを設定しない場合は，デフォルトとしてシードに 1 が用いられる。

互換性——*SYSV* などにも同様の関数があるが，そちらは乱数生成アルゴリズムとして *Linear Congruential Algorithm* と 48 ビット整数演算を用いているため，drand48 という名前である。*LIBC* では異なるアルゴリズムを用いているので，drand48 とはしなかった。

規　格——***Project LIBC Group***

関連項目——rand, random, srand, srandom

dup

用　　途――ファイルハンドルを複製する。

書　　式――#include <unistd.h>

```
int dup (int fildes);
```

解　　説――dup関数は*fildes*で指定したファイルハンドルが指しているファイルに対して，新しいファイルハンドルを割り当てる。複製後はどちらのファイルハンドルを通しても，結果的に同じファイルを操作することになる。

新しく割り当てられるファイルハンドルは，その時点で使用されていないファイルハンドルのなかで最も小さい番号のものとなる。

戻り値――正常に複製できた場合は新しく割り当てられたファイルハンドルを返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　　不正なファイルハンドルを指定した
- EMFILE　　これ以上ファイルハンドルを割り当てられない

規　　格――***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――close，creat，dup2，fcntl，open

dup2

用　　途——ファイルハンドルを複製する。

書　　式——

```
#include <unistd.h>
int dup2 (int fildes1, int fildes2);
```

解　　説——dup2 関数は，*fildes1* と *fildes2* が同じ値を指すように，*fildes2* を新たにオープンする。ただしすでに *fildes2* が使用されていた場合は，複製の前に *fildes2* のファイルハンドルをクローズする。複製後はどちらのファイルハンドルを通しても，結果的に同じファイルを操作することになる。

fildes1 と *fildes2* が同じだった場合は，何も行わずに正常終了する。

戻 り 値——正常に複製できた場合は *fildes2* を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　　不正なファイルハンドルを指定した

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——close，creat，dup，fcntl，open

_enargv

用　途——前処理済みのコマンドラインから引数配列を作成する。

書　式——

```
#include <sys/xstart.h>
int _enargv (const char *prog, int argc,
             char **argv, const char *com);
```

解　説——_enargv 関数は，*com*で指定された前処理(個々の引数に分離)されたコマンドライン文字列と *prog*で指定されたカレントプロセスのファイル名から引数配列を作成し，その結果を *argv*が指す領域に格納する。引数配列は char **型で，引数文字列へのポインタの配列である。

たとえば foobar というプログラムにおいて，*argc*が 4 で *com*が次のようになっていたとすると，

```
arg1\0arg2\0picnic\0this is a pen\0
```

結果として，次のような引数配列が作成される。ここで *argv* [*n*] は，*com*文字列の内部を指している。

```
argv[0] = (char *) "foobar";
argv[1] = (char *) "arg1";
argv[2] = (char *) "arg2";
argv[3] = (char *) "picnic";
argv[4] = (char *) "this is a pen";
```

また，_enargv 関数はライブラリ自身に対するオプションを見つけると，これらを適切に処理し，ユーザプログラムに渡す引数配列には含めないようにする。ライブラリに指定することができるオプションは次のとおりである。

- -+-h:*bytes*　ヒープ領域の初期サイズを *bytes*にする
- -+-s:*bytes*　スタック領域のサイズを *bytes*にする
- -+-p　プログラムをスーパーバイザモードで実行させる
- -+-f　数値計算で FLOAT パッケージを使用するようにする
- -+-g　強制的に C++のグローバルコンストラクタ/デストラクタを起動する

戻 り 値——最終的にユーザプログラムに渡される引数の数，すなわち生成された引数配列の大きさを返す。

注　　意——スタック領域のサイズ，ヒープ領域のサイズチェックは行われないので，不正な値を指定したときの動作は一切保証されない。

規　　格——*Project LIBC Group*

関連項目——`_dehupair`，`_main`，`_start`

_errcnv

用　　途――**Human68k** のエラーコードをライブラリのエラーコードに変換する。

書　　式――`#include <sys/xglob.h>`

```
int _errcnv (int doserrno);
```

解　　説――_errcnv 関数は，*doserrno* で指定した **Human68k** のシステムエラーコードを，ライブラリで使用する**ライブラリエラーコード**に変換する。

戻 り 値――*doserrno* に対応するライブラリエラーコードを返す。

注　　意――システムエラーコードとライブラリエラーコードは 1 対 1 で対応しているわけではないので，必ずしも適切な変換が行われるとはかぎらない。

_errcnv 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格――*Project LIBC Group*

関連項目――perror，strerror

ecvt

E

用　途——浮動小数値を指数形式の数字列に変換する。

書　式——#include <stdlib.h>

```
char *ecvt (double value, int prec, int *dec, int *sign);
```

解　説——ecvt 関数は *value* で指定した浮動小数を指数形式の文字列に変換し，その文字列へのポインタを返す。変換は全体の桁数が *prec* 桁になるように行われ，*value* の符号が *sign* の，小数点が先頭から何バイト目の直前にくるかが *dec* の指す領域に，それぞれ格納される。これらから指数値を求めるには，小数点の位置から 1 を引けばよい。

全体の桁数が *prec* 桁に満たないようならば足りない分だけ "0" を後ろに追加し，桁数が最低でも *prec* 桁になるようにする。たとえば，次のような結果を返す。

```
ecvt (+3.14159000, 3) str = '314' dec =  1 prec = 0
ecvt (+0.31415900, 3) str = '314' dec =  0 prec = 0
ecvt (-0.03141590, 3) str = '314' dec = -1 prec = 1
ecvt (-0.00314159, 3) str = '314' dec = -2 prec = 1
```

戻り値——変換した文字列を格納した関数内部の静的領域へのポインタを返す。

注　意——数値は *prec*+1 桁目で四捨五入によって丸められる。

結果は ecvt 関数内部の静的領域に格納されるため，fcvt 関数や ecvt 関数を呼び出すたびに内容が上書きされることに注意すること。

また printf 関数や sprintf 関数などのフォーマット出力関数は，内部で浮動小数を出力する際に ecvt 関数や fcvt 関数を用いているため，これによって静的領域が破壊されることがある。

互換性——本来ならば printf 関数などのように，ライブラリ自体がライブラリ内部の静的領域を破壊することは避けなければならないが，現在の仕様では上記のような事例が起こり得る。

規　格——***Project LIBC Group***，***XC***，***MS–C 7.0***

関連項目——fcvt，gcvt

endgrent

用　　途――グループファイルへのアクセスを終了する。

書　　式――#include <grp.h>

void endgrent (void);

解　　説――endgrent 関数は，setgrent 関数や getgrent 関数などでアクセス中のグループファイルへのアクセスを終了する。

getgrnam 関数や getgrgid 関数のインタフェイスを用いずにグループを検索する場合は，明示的に endgrent 関数によってアクセスを終了すること。

戻り値――なし。

規　　格――***Project LIBC Group*, *4.3BSD*, *SYSV***

関連項目――getgrent, getgrgid, getgrnam, setgrent

endpwent

用　途——パスワードファイルへのアクセスを終了する。

書　式——

```
#include <pwd.h>
void endpwent (void);
```

解　説——endpwent 関数は，setpwent 関数や getpwent 関数などでアクセス中のパスワードファイルへのアクセスを終了する。

getpwnam 関数や getpwuid 関数のインタフェイスを用いずにグループを検索する場合は，明示的に endpwent 関数によってアクセスを終了すること。

戻り値——なし。

規　格——*Project LIBC Group*，*4.3BSD*，*SYSV*

関連項目——getpwent，getpwnam，getpwuid，setpwent

environ

用　途——環境変数テーブル。

書　式——
```
#include <unistd.h>
extern char **environ;
```

解　説——変数 environ は環境変数テーブルであり，個々の環境変数へのポインタの配列である。個々の環境変数とは"*name=value*"というフォーマットの文字列であり，1 環境変数が 1 文字列で表現されている。

注　意——ここでいう環境変数テーブルとは，プロセスの内部にコピーされた環境のことを指し，親プロセスや **Human68k** の環境変数テーブルのことではない。また変数 environ は，putenv 関数や clearenv 関数の実行によって変化することがあるため，その値をコピーしたポインタは必ずしもテーブルへのアクセスを保証されない。

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——clearenv，exec，getenv，putenv，spawn

eprintf

E

用　途——標準エラー出力ファイルストリームにフォーマット出力を行う。

書　式——

```
#include <stdio.h>
int eprintf (const char *format, ...);
```

解　説——eprintf 関数は，*format*で指定したフォーマット文字列にしたがって引数を文字列に変換し，結果を**標準エラー出力**に割り当てられた**ファイルストリーム**に出力する。

フォーマット文字列の指定の方法や詳細な説明については，fprintf 関数を参照のこと。

戻り値——正常に出力できた場合は出力した文字数を返し，失敗した場合は EOF を返してストリームの**エラー指示子**を設定する。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　格——*Project LIBC Group*

関連項目——fprintf，printf，sprintf

execl

用　途――プログラムを実行する。

書　式――

```
#include <unistd.h>
int execl (const char *path, const char *arg0, ...);
```

解　説――execl 関数は現在実行中のプログラムに代わり，*path*で指定したプログラムを起動して制御を移す。したがって execl 関数が正常に処理された場合，execl 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execl 関数には *arg*以降に，任意の数の引数列（文字列の集合）を渡すことができ，起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv* という引数配列が渡されるが，*arg*0 からの一連の引数は，この *argv*配列を直接指定していることと同じである。ただし，*argv*[0] はプログラム自身を表すので，普通は *path* と同じ値を設定する。また引数列の最後に，終端記号として NULL を指定すること。

戻り値――正常に実行できた場合は execl 関数は戻ってこないが，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意――実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互換性――**Human68k** には **UNIX** の exec 関数と同等の概念がないので，*LIBC*では spawn 関数を流用して，exec 関数を疑似的に作り出している。起動するプログラムはつねに**子プロセス**として起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上は exec 関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らなければならず，親プロセスが atexit 関数や onexit 関数で設定した後，処理ルーチンが呼び出されてしまう。

規　格――***Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目――execle，execlp，execv，execve，execvp

execle

用　途——プログラムを実行する。

書　式——

```
#include <unistd.h>
int execle (const char *path, const char *arg0, ...
            char *const envp[]);
```

E

解　説——execle 関数は現在実行中のプログラムに代わり，*path* で指定したプログラムを起動して制御を移す。したがって，execle 関数が正常に処理された場合，execle 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execle 関数には *arg* 以降に，任意の数の引数列（文字列の集合）を渡すことができ，起動されるプログラムに引数として渡される。C のプログラムは起動されると，main 関数に *argc* と *argv* という引数配列が渡されるが，*arg*0 からの一連の引数は，この *argv* 配列を直接指定しているのと同じである。ただし *argv*[0] はプログラム自身を表すため，普通は *path* と同じ値を設定する。また引数列の最後に，終端記号として NULL を指定すること。

execle 関数は実行するプログラムに与える環境エリアを，*envp* の配列で指定することができる（*envp* は引数列の最後に設定される NULL の次である）。実行されるプログラムが，もし getenv 関数などによって環境変数を検索しようとすると，この配列の内容が検索されることになる。この配列の構造については変数 environ を参照のこと。

戻り値——正常に実行できた場合は execle 関数は戻ってこないが，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互 換 性——Human68kにはUNIXのexec関数と同等の概念がないので，*LIBC*ではspawn関数を流用して，exec関数を疑似的に作り出している。起動するプログラムはつねに子プロセスとして起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上はexec関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らねばならず，親プロセスがatexit関数やonexit関数で設定した後，処理ルーチンが呼び出されてしまう。

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——execl，execlp，execv，execve，execvp

execlp

用　途——プログラムを実行する。

書　式——

```
#include <unistd.h>
int execlp (const char *file, const char *arg0, ...);
```

解　説——execlp 関数は現在実行中のプログラムに代わり，*file*で指定したプログラムを起動して制御を移す。したがって execlp 関数が正常に処理された場合，execlp 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execlp 関数には *arg* 以降に，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv* という引数配列が渡されるが，*arg*0 からの一連の引数は，この *argv* 配列を直接指定しているのと同じである。ただし，*argv* [0] はプログラム自身を表すため，普通は *file* と同じ値を設定する。また引数列の最後に，終端記号として NULL を指定すること。

呼び出されるプログラムは *file* で指定する。もし *file* が "/" や "\" のようなパスの区切り記号を含んでいれば，*file* はプログラムのパスを示しているとみなされる。しかし区切り記号が含まれていなければ，execlp 関数は *file* という名前のファイルを環境変数 path に設定されているディレクトリのなかから検索し，最初に見つかったものを実行する。

戻り値——正常に実行できた場合は execlp 関数は戻ってこないが，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*file* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互 換 性——**Human68k**には**UNIX**のexec関数と同等の概念がないので，*LIBC*ではspawn関数を流用して，exec関数を疑似的に作り出している。起動するプログラムはつねに子プロセスとして起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上はexec関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らねばならず，親プロセスがatexit関数やonexit関数で設定した後，処理ルーチンが呼び出されてしまう。

規　　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——execl，execle，execv，execve，execvp

execv

用　途——プログラムを実行する。

書　式——#include <unistd.h>

```
int execv (const char *path, char *const argv[]);
```

解　説——execv 関数は現在実行中のプログラムに代わり，*path* で指定したプログラムを起動して制御を移す。したがって execv 関数が正常に処理された場合，execv 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execv 関数には *argv* で任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv* という引数配列が渡されるが，execv 関数で指定する *argv* は，この main 関数の *argv* 配列を直接指定していることと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は *path* と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

戻り値——正常に実行できた場合は execv 関数は戻ってこないが，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互換性——**Human68k** には **UNIX** の exec 関数と同等の概念がないため，*LIBC* では spawn 関数を流用して exec 関数を疑似的に作り出している。起動するプログラムはつねに**子プロセス**として起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上は exec 関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らねばならず，親プロセスが atexit 関数や onexit 関数で設定した後，処理ルーチンが呼び出されてしまう。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——execl，execle，execlp，execve，execvp

execve

用　途――プログラムを実行する。

書　式――#include <unistd.h>

```
int execve (const char *path, char *const argv[]
            char *const envp[]);
```

解　説――execve 関数は現在実行中のプログラムに代わり，*path*で指定したプログラムを起動して制御を移す。したがって execve 関数が正常に処理された場合，execve 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execve 関数には*argv*で任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。Cのプログラムは起動時されると main 関数に *argc* と *argv*という引数配列が渡されるが，execve 関数で指定する *argv*は，この main 関数の *argv*配列を直接指定していることと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は *path* と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

execve 関数は実行するプログラムに与える環境エリアを，*envp*の配列で指定することができる (*envp*は引数列の最後に設定される NULL の次である)。実行されるプログラムが，もし getenv 関数などによって環境変数を検索しようとすると，この配列の内容が検索されることになる。この配列の構造については変数 environ を参照のこと。

戻り値――正常に実行できた場合は execve 関数は戻ってこないが，失敗した場合には−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path*で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意――実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互 換 性——Human68k には UNIX の exec 関数と同等の概念がないので，*LIBC* では spawn 関数を流用して，exec 関数を疑似的に作り出している。起動するプログラムはつねに**子プロセス**として起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上は exec 関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らねばならず，親プロセスが atexit 関数や onexit 関数で設定した後，処理ルーチンが呼び出されてしまう。

規　　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——execl，execle，execlp，execv，execvp

execvp

用　途――プログラムを実行する。

書　式――

```
#include <unistd.h>

int execvp (const char *file, char *const argv[]);
```

解　説――execvp 関数は現在実行中のプログラムに代わり，*file* で指定したプログラムを起動して制御を移す。したがって execvp 関数が正常に処理された場合，execvp 関数は戻らず，また現在実行中のプログラムもこれ以降処理されない。

execvp 関数には *argv* で任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv* という引数配列が渡されるが，execvp 関数で指定する *argv* は，この main 関数の *argv* 配列を直接指定しているのと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は *file* と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

呼び出されるプログラムは *file* で指定する。もし *file* が "/" や "\" のようなパスの区切り記号を含んでいれば，*file* はプログラムのパスを示しているとみなされる。しかし区切り記号が含まれていなければ，execvp 関数は *file* という名前のファイルを環境変数 path に設定されているディレクトリのなかから検索し，最初に見つかったものを実行する。

戻り値――正常に実行できた場合は execvp 関数は戻ってこないが，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*file* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，ループしている
- ENOMEM　メモリが足りなくなった

注　意――実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

互 換 性——Human68k には UNIX の exec 関数と同等の概念がないので，*LIBC* では spawn 関数を流用して，exec 関数を疑似的に作り出している。起動するプログラムはつねに**子プロセス**として起動されるが，その子プロセスが終了すると親プロセスも即座に終了するため，見かけ上は exec 関数と同じになる。ただし上記のような理由で，内部的には一度親プロセスへ戻らねばならず，親プロセスが atexit 関数や onexit 関数で設定した後，処理ルーチンが呼び出されてしまう。

規　　格——*Project LIBC Group*， *POSIX.1*， *XPG3*， *AES/OS*， *4.3BSD*， *SYSV*

関連項目——execl，execle，execlp，execv，execve

exit

用　　途——プロセスを終了させる。

書　　式——#include <stdlib.h>

```
void __volatile exit (int exitcode);
```

解　　説——exit 関数は現在実行中のプロセスを終了し，その終了コードとして *exitcode* を指定する。

exit 関数はまず atexit 関数で登録された関数を順番に実行し，次にオープンされていたすべてのファイルハンドル，ファイルストリームをクローズする。そして tmpfile 関数でオープンしたテンポラリファイルを削除し，C++のグローバルデストラクタを起動する。

戻 り 値——exit 関数は決して呼び出し側には戻ってこない。

注　　意——オープンファイルのクローズやテンポラリファイルの削除，C++のグローバルデストラクタの起動は，実際には exit 関数から呼び出される _exit 関数によって行われる。

規　　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——atexit，close，fclose，tmpfile

exp

用　途──e^xを求める。

書　式──
```
#include <math.h>
double exp (double x);
#include <sys/xmath.h>
double _f_exp (double x);
double _fe_exp (double x);
double _fpu_exp (double x);
```

解　説──exp 関数は，ネイピア数 e を底とする指数 x の累乗 e^x を求める。各関数はそれぞれ次のように動作する。

- exp　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_exp　数値演算コプロセッサ命令を直接ドライブする
- _fe_exp　FLOAT パッケージを呼び出す
- _fpu_exp　I/O 数値演算コプロセッサを直接ドライブする

戻り値──正しく値が求められた場合，exp 関数は $x > 0$ の範囲で x の指数を返す。もしも x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返し，それ以外では無限大を返す。

- EDOM　x の値が非数
- ERANGE　計算結果がオーバーフロー，またはアンダーフロー

注　意──<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは exp は _fe_exp の別名となり，__DIRECT_IOFPU__が定義されているときは exp は _fpu_exp の別名となる。また，__DIRECT_FPU__が定義されているときは exp は _f_exp の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　格──*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目──isnan，log，log10，pow

_fpu_off

用　　途――数学ライブラリをFLOATパッケージ呼び出しにする。

書　　式――#include <sys/xmath.h>

```
void _fpu_off (void);
```

解　　説――_fpu_off関数は数値演算コプロセッサの有無にかかわらず，これ以降に呼び出される数学ライブラリ関数をFLOATパッケージ呼び出しで動作させる。***LIBC***は，起動時に数値演算コプロセッサの有無に応じて演算ルーチンを切り替えるようになっているため，どうしてもFLOATパッケージを使用したい場合には_fpu_off関数を用いなければならない。

戻り値――なし。

注　　意――_fpu_off関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格――***Project LIBC Group***

関連項目――_fpu_on

_fpu_on

用　途——数学ライブラリを数値演算コプロセッサ直接駆動にする。

書　式——

```
#include <sys/xmath.h>
void _fpu_on (void);
```

解　説——_fpu_on 関数は数値演算コプロセッサが使用できる場合，これ以降に呼び出される数学ライブラリ関数を数値演算コプロセッサ直接駆動で動作させる。*LIBC*は，起動時に数値演算コプロセッサの有無に応じて演算ルーチンを切り替えるようになっている。そのため，明示的に FLOAT パッケージおよび数値演算コプロセッサを指定するには，_fpu_on 関数を用いなければばならない。

戻り値——なし。

注　意——_fpu_on 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——*Project LIBC Group*

関連項目——_fpu_off

F

_fullentry

用　途——ファイル名をフルパスに展開する。

書　式——#include <sys/xstdlib.h>

```
char *_fullentry (char *dst, const char *src, size_t maxlen);
```

解　説——_fullentry 関数は *src* で指定したファイル名をフルパスに展開し，*dst* の指す *maxlen* バイトの領域にコピーする。_fullentry 関数は _fullpath 関数とは異なり，ディレクトリエントリの展開は行わない。

foo/. → A:/foo/.
foo/.. → A:/foo/..

仮想ドライブのファイル名については，物理ドライブのフルパスに展開する。たとえば，ディレクトリ A:/foo がドライブ V: に割り当てられていた場合，次のようになる。

V:/file → A:/foo/file

なお，仮想ディレクトリのファイル名は物理ドライブが存在しなくなるため，通常ファイルと同等に展開する。

戻り値——正常に展開できた場合は *dst* を返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ERANGE　展開したファイル名を格納する領域が足りない
- ENOENT　不正なドライブを指定した

注　意——_fullentry 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——_dos_getassign，_dos_nameck，_fullpath

_fullpath

用　　途——ファイル名をフルパスに展開する。

書　　式——#include <stdlib.h>

```
char *_fullpath (char *dst, const char *src, size_t maxlen);
```

解　　説——_fullpath 関数は *src* で指定されたファイル名をフルパスに展開し，*dst* の指す *maxlen* バイトの領域にコピーする。_fullpath 関数は _fullentry 関数とは異なり，ディレクトリエントリの展開も行う。

```
foo/.  → A:/foo
foo/.. → A:/
```

仮想ドライブのファイル名については，物理ドライブのフルパスに展開する。たとえば，ディレクトリ A:/foo がドライブ V: に割り当てられていた場合，次のようになる。

```
V:/file → A:/foo/file
```

なお仮想ディレクトリのファイル名は，物理ドライブが存在しなくなるため通常ファイルと同等に展開する。

戻 り 値——正常に展開できた場合は *dst* を返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ERANGE　展開したファイル名を格納する領域が足りない
- ENOENT　不正なドライブを指定した

規　　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——_dos_getassign，_dos_nameck，_fullentry

fabs

用　途——x の絶対値を返す。

書　式——#include <math.h>

```
double fabs (double x);
#include <sys/xmath.h>
double _f_fabs (double x);
double _fe_fabs (double x);
double _fpu_fabs (double x);
```

解　説——fabs 関数は x の値の絶対値を返す。各関数はそれぞれ次のように動作する。

- fabs　　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_fabs　　数値演算コプロセッサ命令を直接ドライブする
- _fe_fabs　　FLOAT パッケージを呼び出す
- _fpu_fabs　　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合はその値を返す。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して，同じく非数を返す。

- EDOM　　x の値が非数

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは fabs は _fe_fabs の別名となり，__DIRECT_IOFPU__が定義されているときは fabs は _fpu_fabs の別名となる。また，__DIRECT_FPU__が定義されているときは fabs は _f_fabs の別名となる。

規　格——***Project LIBC Group，ANSI C，POSIX.1，XPG3，AES/OS，4.3BSD，SYSV***

関連項目——ceil，floor，fmod

fchmod

用　途――ファイルのアクセスモードを変更する。

書　式――#include <stat.h>

```
int fchmod (int fildes, mode_t mode);
```

解　説――fchmod 関数は，*fildes* で指定したファイルハンドルが指すファイルのファイルモードを *mode* で指定する値に変更する。*mode* に指定することができるのは次のとおりで，それぞれの値の論理和を指定する。

- S_IEXEC　実行ファイル，チェンジディレクトリ可能
- S_IWRITE　書き込み可
- S_IREAD　読み込み可

戻り値――正常にアクセスモードの変更ができた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- EPERM　ボリュームファイルを変更しようとした

互換性――**Human68k** ではファイルに実行属性や読み込み禁止属性を設定することはできないので（実行属性については execd を使えば可能），S_IEXEC や S_IREAD は疑似的に作り出される値である。

規　格――***Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目――chmod，fstat，_mode2dos，stat

fchown

用　　途――ファイルのオーナーおよびグループを変更する。

書　　式――#include <unistd.h>

```
int fchown (int fildes, uid_t owner, gid_t group);
```

解　　説――fchown関数は，*fildes*で指定したファイルハンドルが指すファイルのオーナーIDおよびグループIDを変更する。

戻 り 値――正常に変更できた場合は0を返し，失敗した場合には-1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　不正な*fildes*を指定した
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

互 換 性――**Human68k**にはユーザIDやグループIDの概念がないので，fchown関数がこれらの値に影響されることはない。また，***LIBC***内ではデフォルトの仮想ユーザIDおよび仮想グループIDをrootと仮定しているため，fchown関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――chmod，chown，fchmod

fclose

用　途——ファイルストリームをクローズする。

書　式——#include <stdio.h>

```
int fclose (FILE *stream);
```

解　説——fclose 関数は *stream* が指すファイルストリームをクローズする。まず fclose 関数は，バッファリングされていた未書き込みデータをすべて書き出し，読み込みデータは捨てる。次に，バッファとして自動的に割り当てられていたメモリを解放し，最後にオープンしていたファイルをクローズする。

戻り値——正常にクローズできた場合は 0 を返し，失敗した場合は EOF を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連していたファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　意——ユーザが setvbuf 関数を用いて割り当てたバッファは自動的には解放されない。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——close，fflush，fopen

fcloseall

用　途——すべてのファイルストリームをクローズする。

書　式——
```
#include <stdio.h>
int fcloseall (void);
```

解　説——fcloseall関数は，現在オープンされているすべてのファイルストリームをクローズする。詳細はfclose関数を参照のこと。

戻り値——正常にクローズできた場合は0を返し，1つでも失敗した場合はEOFを返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連していたファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——fclose，fflush

fcntl

用　　途——ファイルとファイルハンドルの操作を行う。

書　　式——#include <fcntl.h>

```
int fcntl (int fildes, int cmd, ...);
```

解　　説——fcntl 関数は *fildes* で指定されたファイルハンドルが指すファイルを操作する機能を提供する。どういう操作を行うかは *cmd* で指定することができ，以下のコマンドを指定することができる。

- F_DUPFD　第３の引数である *arg* を取り出し，*arg* と等しいかそれ以上の番号をもつ未使用のファイルハンドルを割り当てて，そのファイルハンドルが *fildes* の指すファイルと同じファイルを指すようにする。新しいファイルハンドルは，*arg* 以上のもののうち小さい番号から順に空いているものが割り当てられる。動作としては dup2 関数と類似している。
- F_GETFL　*fildes* で指定したファイルハンドルに設定されているファイルステータスフラグ (O_APPEND) と，ファイルアクセスフラグ (O_RDONLY，O_WRONLY，O_RDWR) を返す。
- F_SETFL　*fildes* で指定したファイルハンドルに設定されているファイルステータスフラグ (O_APPEND) を第３の引数である *arg* で再設定する。ファイルアクセスフラグ (O_RDONLY，O_WRONLY，O_RDWR) は変化しない。

戻 り 値——正常にファイルコントロールできた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- EINVAL　不正な *cmd* を指定した
- EMFILE　これ以上ファイルハンドルを割り当てられないか，F_DUPFD で第３の引数 *arg* 以上の番号で空いているファイルハンドルがない

互 換 性——fcntl 関数のもつ他の多くの機能，たとえばファイルディスクリプタフラグの取得，再設定，ロックの設定，再設定などは省略されている。

規　　格——***Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——close，open

fcvt

用　　途──浮動小数値を数字列に変換する。

書　　式──#include <stdlib.h>

```
char *fcvt (double value, int prec, int *dec, int *sign);
```

解　　説──fcvt 関数は *value* で指定した浮動小数を文字列に変換し，その文字列へのポインタを返す。変換は小数点以下の桁数が *prec* 桁になるように行われ，*sign* には *value* の符号が，*dec* の指す領域には，変換された文字列の何バイト目に小数点がくるのかが格納される。つまり，小数点が $dec-1$ バイト目と *dec* バイト目の間にかくされていることになる。

ただし全体の桁数が *prec* 桁に満たないようならば，足りない分だけ "0" を後ろに追加し，桁数が最低でも *prec* 桁になるようにする。たとえば，次のような結果を返す。

```
fcvt (+3.14159000, 3) str = '3142' dec =  1 sign = 0
fcvt (+0.31415900, 3) str = '314'  dec =  0 sign = 0
fcvt (-0.03141590, 3) str = '310'  dec = -1 sign = 1
fcvt (-0.00314159, 3) str = '300'  dec = -2 sign = 1
```

戻 り 値──変換した文字列を格納した関数内部の静的領域へのポインタを返す。

注　　意──数値は小数点以下 *prec*+1 桁目で四捨五入によって丸められる。

結果は fcvt 関数内部の静的領域に格納されるため，fcvt 関数や ecvt 関数を呼び出すたびに内容が上書きされることに注意すること。

また，printf 関数や sprintf 関数などのフォーマット出力関数は，内部で浮動小数を出力する際に fcvt 関数や ecvt 関数を用いているために，静的領域が破壊されることがある。

互 換 性──本来ならば printf 関数などのように，ライブラリ自体がライブラリ内部の静的領域を破壊することは避けなければならないが，現在の仕様では上記のような事例が起こり得る。

規　　格──***Project LIBC Group***，***XC***，***MS–C 7.0***

関連項目──ecvt，gcvt

fdopen

用　途——ファイルハンドルに対するファイルストリームをオープンする。

書　式——

```
#include <stdio.h>
FILE *fdopen (int fildes, const char *mode);
```

解　説——fdopen 関数は，*fildes* で指定したファイルハンドルに対するファイルストリームをオープンする。*mode* には fopen 関数と同様にストリームのモードを指定することができ，次のなかから1つを選択することができる。ただし，選択は *fildes* が指すファイルの**オープンモード**で許される範囲に限られる。

- "r"　読み込み
- "w"　書き込み
- "a"　追加書き込み
- "r+"　更新 (読み込みと書き込み)
- "w+"　更新 (読み込みと書き込み)
- "a+"　更新 (読み込みと追加書き込み)

ただし fdopen 関数は fopen 関数とは異なり，"w" や "w+" を指定しても関連するファイルのサイズを 0 バイトに戻したりはしない。また，モードを表す文字列の最後に次の文字を指定することで，ストリームのモードをテキスト/バイナリのどちらかにすることができる。省略時は *fildes* が指すファイルのオープンモードに合わせられる。

- "b"　バイナリモード
- "t"　テキストモード (CRLF → LF 変換)

"r+b" → バイナリモードでの更新

戻り値——正常にオープンできた場合はストリームへのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　*fildes* で指定したファイルハンドルが不正である
- EMFILE　これ以上ファイルストリームを割り当てられない
- EINVAL　*mode* で不正なモードを指定した
- ENOMEM　メモリが足りなくなった
- EACCES　*mode* で指定したモードは許されない

規　格——*Project LIBC Group*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——fclose, fopen, freopen, open

feof

用　　途——ファイルストリームの終端指示子を調べる。

書　　式——

```
#include <stdio.h>
int feof (FILE *stream);
```

解　　説——feof 関数は *stream* が指すファイルストリームの終端指示子の状態を調べる。終端指示子がセットされるのは，読み込み中にファイルの終端に達した場合である。

戻 り 値——ファイルストリームに終端指示子がセットされていれば 0 以外の値を返し，セットされていなければ 0 を返す。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——clearerr，ferror

ferror

用　途——ファイルストリームのエラー指示子を調べる。

書　式——
```
#include <stdio.h>
int ferror (FILE *stream);
```

解　説——ferror関数は*stream*が指すファイルストリームのエラー指示子の状態を調べる。エラー指示子がセットされるのは，対象となるファイルストリームでのデータ入出力中に何らかのエラーが発生した場合や，正しい操作を行わずに読み込みから書き込みへ，またはその逆の操作を行った場合である。

戻り値——ファイルストリームにエラー指示子がセットされていれば0以外の値を，セットされていなければ0を返す。

注　意——ファイルストリームにエラー指示子が設定されている場合，変数errnoにその原因を示すエラーコードを設定する。ただし変数errnoの値は，ファイル操作の直後しか保証されない。

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——clearerr，feof

fflush

用　途——ファイルストリームをフラッシュする。

書　式——#include <stdio.h>

```
int fflush (FILE *stream);
```

解　説——*stream*が指す**ファイルストリーム**が書き込みモード，あるいは更新モードで最後に行った操作が読み込みではなかった場合，fflush関数はバッファ上に残った未書き込みのデータをフラッシュし，実際にデータを出力する。

*stream*が指すファイルストリームが読み込みモード，あるいは更新モードで最後に行った操作が書き込みではなかった場合，fflush関数は前回バッファ上から読み込んだデータの次にシステムの**ファイルポインタ**がくるように補正する。ただしこれは，ファイルストリームに割り付けられたファイルがシーク可能な場合に限られる。

戻り値——正常にフラッシュできた場合は0を返し，失敗した場合はEOFを返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　意——ファイルストリームが**テキストモード**でオープンされている場合，ファイルポインタの補正が正しく行われるという保証はない。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——close，exit，fdopen，fopen，fread，fwrite

ffs

用　　途——セットされたビットを検索する。

書　　式——#include <string.h>

```
int ffs (int bitarray);
```

解　　説——ffs関数は*bitarray*のなかでセットされた最初のビットを見つけ，そのビットの桁位置(インデックス)を返す。ビットは1から数える。

戻 り 値——ffs関数は最初に見つけたビットの桁位置を返す。しかし*bitarray*が0で，ビットが見つからない場合は0を返す。

規　　格——***4.3BSD***

関連項目——bcmp，bcopy，bzero

fgetc

用　途——ファイルストリームから1バイトを取り出す。

書　式——
```
#include <stdio.h>
int fgetc (FILE *stream);
```

解　説——fgetc 関数は，*stream* で指定された**ファイルストリーム**から unsigned char 型のデータとして1バイトを取り出し，int 型に変換して返す。ストリームのファイル**ポインタ**はこれに応じて1バイト分進む。

戻り値——正常に取り出せた場合は，そのデータを返す。もしストリーム操作中に何らかのエラーが発生した場合はストリームの**エラー指示子**を設定して EOF を返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　　ファイルストリームに関連するファイルハンドルがおかしい

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——getc

fgetpos

用　途——ファイルストリームのファイルポインタの位置を取得する。

書　式——#include <stdio.h>

```
int fgetpos (FILE *stream, fpos_t *pos);
```

解　説——fgetpos関数は*stream*で指定されたファイルストリームの現在のファイルポインタの位置を求め，結果を*pos*が指す領域に格納する。この値は，fsetpos関数を用いてファイルポインタの位置を再設定する場合に使用することができる。

戻り値——正しくファイルポインタの位置を得ることができれば0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- ESPIPE　ファイルストリームに関連するファイルはシークできない

注　意——ファイルストリームがテキストモードでオープンされている場合は，fgetpos関数が正しい値を返す保証はない。同様に，fsetpos関数で位置を再設定しても元の位置に戻る保証はない。

規　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——fopen，fseek，fsetpos，rewind

fgets

用　途——ファイルストリームから文字列を取り出す。

書　式——#include <stdio.h>

```
char *fgets (char *s, int n, FILE *stream);
```

解　説——fgets 関数は *stream* で指定されたファイルストリームから文字列を取り出し，結果を *s* が指す領域に格納する。文字列の取り出しは $n-1$ バイトを処理するか，ファイルの終端に到達するか，**改行文字**を処理するまで続けられる。結果の文字列には改行文字が含まれ，最後に null 文字が置かれることに注意する。

戻り値——正しく文字列を取り出すことができた場合には *s* を返す。もしストリーム操作中に何らかのエラーが発生した場合は，ストリームの**エラー指示子**を設定して NULL を返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定して NULL を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——*s* は少なくとも *n* バイトの領域を指していなければならない。

規　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——ferror，fread，gets

filelength

用　　途——ファイルの長さを求める。

書　　式——
```
#include <unistd.h>
int filelength (int fildes);
```

解　　説——filelength関数は*fildes*で指定したファイルハンドルが指すファイルの長さを求める。

戻 り 値——正常に長さを求められた場合はその長さをバイト単位で返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　　不正な*fildes*を指定した

規　　格——*Project LIBC Group*，*XC*，*MS–C 7.0*

関連項目——chsize，fileno，fstat

fileno

用　　途——ファイルストリームに関連するファイルハンドルを取得する。

書　　式——

```
#include <stdio.h>
int fileno (FILE *stream);
```

解　　説——fileno 関数は *stream* で指定したファイルストリームに関連するファイルハンドルを返す。

戻 り 値——成功した場合はファイルハンドルを返し，失敗した場合は−1 を返す。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——fdopen，fopen

floor

用　　途――x 以下の整数のなかで最も大きな数を返す。

書　　式――#include <math.h>

```
#include <math.h>
double floor (double x);
```

解　　説――floor 関数は x 以下の値をもつ整数のなかで最も大きな数を返す。

戻 り 値――正しく値が求められた場合はその値を返す。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　x の値が非数，または±HUGE_VAL だった

注　　意――つねに FLOAT パッケージを呼び出す。

規　　格――*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――ceil

flushall

用　　途――すべてのファイルストリームをフラッシュする。

書　　式――#include <stdio.h>

```
int flushall (void);
```

解　　説――flushall 関数は現在オープンされているすべてのファイルストリームをフラッシュする。詳細は fflush 関数を参照のこと。

戻 り 値――正常にフラッシュできた場合は 0 を返し，1 つでも失敗した場合は EOF を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

規　　格――*Project LIBC Group,　MS-C 7.0*

関連項目――fflush

fmod

用　途 —— x/y の剰余を返す。

書　式 ——

```
#include <math.h>
double fmod (double x, double y);
#include <sys/xmath.h>
double _f_fmod (double x, double y);
double _fe_fmod (double x, double y);
double _fpu_fmod (double x, double y);
```

解　説 —— fmod関数はx/yの浮動小数点値の剰余を返す。各関数はそれぞれ次のように動作する。

- fmod　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合はFLOATパッケージを呼び出す
- _f_fmod　数値演算コプロセッサ命令を直接ドライブする
- _fe_fmod　FLOATパッケージを呼び出す
- _fpu_fmod　I/O数値演算コプロセッサを直接ドライブする

戻り値 —— 正しく値が求められた場合はその値を返す。もし x または y が非数(NaN)ならば，変数errnoにその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　x または y の値が非数，または y の値が0

注　意 —— <math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときはfmodは_fe_fmodの別名となり，__DIRECT_IOFPU__が定義されているときはfmodは_fpu_fmodの別名となる。また，__DIRECT_FPU__が定義されているときはfmodは_f_fmodの別名となる。

規　格 —— *Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目 —— ceil，fabs，floor

fmode

用　途——ファイルストリームの変換モードを変更する。

書　式——#include <stdio.h>

```
void fmode (FILE *stream, int mode);
```

解　説——fmode 関数は *stream* で指定したファイルストリームのテキスト/バイナリ変換モードを *mode* に変更する。*mode* には次の値を指定することができる。

- _IOTEXT　ファイルストリームの変換モードをテキストモードに変更する
- _IOBINARY　ファイルストリームの変換モードをバイナリモードに変更する

戻り値——なし。

規　格——*Project LIBC Group*, *XC*

関連項目——fdopen, fopen, freopen, setmode

fopen

用　　途――ファイルストリームをオープンする。

書　　式――#include <stdio.h>

```
FILE *fopen (const char *path, const char *mode);
```

解　　説――fopen関数は*path*で指定したファイルをオープンし，それに対応するファイルストリームを作成する。*mode*にはストリームのモードを指定することができ，次のなかから1つを選択することができる。ただし，選択はファイルのファイルアクセスモードで許される範囲に限られる。

- "r"　読み込み
- "w"　書き込み
- "a"　追加書き込み
- "r+"　更新(読み込みと書き込み)
- "w+"　更新(読み込みと書き込み)
- "a+"　更新(読み込みと追加書き込み)

ファイルストリームが"w"や"w+"を指定して書き込み可能な形でオープンした場合，すでに*path*で指定したファイルが存在するならばその内容は破棄され，そのサイズは0バイトに切り詰められる。

また，モードを表す文字列の最後に次の文字を指定することで，ストリームのモードをテキスト/バイナリのどちらかにすることができる。省略時はテキストモード，あるいはfmode関数で指定したモードに合わせられる。

- "b"　バイナリモード
- "t"　テキストモード(CRLF → LF変換)

"r+b" → バイナリモードでの更新

戻り値――正常にオープンできた場合はストリームへのポインタを返し，失敗した場合はNULLを返して，変数errnoにその原因を示すエラーコードを設定する。

- EISDIR　ディレクトリである
- ENOSPC　ディスクがいっぱいである
- EROFS　リードオンリーファイルシステムである
- EMFILE　これ以上ファイルストリームを割り当てられない
- EINVAL　*mode*で不正なモードを指定した

- ENOMEM　メモリが足りなくなった
- EACCES　*mode*で指定したモードは許されない

規　格——*Project LIBC Group*，　*ANSI C*，　*POSIX.1*，　*XPG3*，　*AES/OS*，　*4.3BSD*，*SYSV*

関連項目——fclose，fdopen，freopen，open

F

fpathconf

用　途——パス名に関する情報を取り出す。

書　式——

```
#include <unistd.h>
long fpathconf (int fildes, int name);
```

解　説——fpathconf 関数は *fildes* で指定したファイルハンドルに関連するパスの情報を取得し，ユーザアプリケーションに対して現在のシステム設定値などを得る手段を与える。どの設定値を調べるかは *name* で指定する。*name* に指定することができるマクロ値と求める設定値は次のとおり。

- `_PC_LINK_MAX`　ハードリンクの最大値 (参考値)
- `_PC_MAX_CANNON`　端末の Canonical 入力の最大バイト数 (参考値)。*path* は端末デバイスを指していなければならない
- `_PC_MAX_INPUT`　端末の入力キューの最大バイト数 (参考値)。*path* は端末デバイスを指していなければならない
- `_PC_NAME_MAX`　ファイル名に許される最大の長さ (ただし null 文字は含まない)
- `_PC_PATH_MAX`　パス名に許される最大の長さ (ただし null 文字は含まない)
- `_PC_PIPE_BUF`　パイプバッファの最大バイト数 (*LIBC* では疑似パイプしかサポートしていないので，参考値にすぎない)
- `_PC_CHOWN_RESTRICTED`　chown 関数の使用が制限されているかどうか (*LIBC* では chown 関数が必ず成功するので無制限)
- `_PC_NO_TRUNC`　`NAME_MAX` よりも長いファイル名を，適正な長さまで切り詰めるかどうか (*LIBC* では一切考慮していないので，切り詰めはない)
- `_PC_VDISABLE`　端末の特殊文字を無効にすることができる文字 (参考値)。*path* は端末デバイスを指していなければならない

戻り値——正常に制限値を取得できた場合はその値 (無制限ならば-1) を返し，失敗した場合には-1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- `EINVAL`　不正な *name* を指定したか，または *name* に関する制限値が求められない
- `EBADF`　不正なファイルハンドルを指定した

- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

互 換 性—— *LIBC*では上記 (詳細は<limits.h>を参照) の制限値は可変ではなく，すべて固定である。

規　　格—— ***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——pathconf，sysconf

F

fprintf

用　途――指定したファイルストリームにフォーマット出力を行う。

書　式――#include <stdio.h>

```
int fprintf (FILE *stream, const char *format, ...);
```

解　説――fprintf 関数は，*format* で指定したフォーマット文字列にしたがって引数を文字列に変換し，結果を *stream* が指すファイルストリームに出力する。

format に指定するフォーマット文字は通常の文字列であるが，“%” で表される**変換指定文字**に続く一連の変換方法の書式指定があると，その指示にしたがって引数が文字列に変換される。これらの変換 (たとえば小数点の文字など) は，現在の**ロケール**の LC_NUMERIC カテゴリに左右される。

通常，変換の書式指定は次のような形式である。

%[フラグ][最小フィールド幅][精度][変換修飾子][変換指定子]

1. フラグは任意の命令の変換仕様を修飾するもので，0 個以上の任意の数を指定することができ，次のようなものがある。

   - \- 左詰めフラグ
     変換結果をフィールド内部で左詰めする。このフラグを指定しない場合は右詰めとなる
   - \+ 符合フラグ
     数値を符合つき変換 (d, i, f, e, E, g, G) した場合，値が正ならば頭に “+” を付加し，つねに符合がつくようにする。このフラグを指定しない場合は，負の値に限り符合がつく
   - ‘ ’ 空白フラグ
     数値を符合つき変換 (d, i, f, e, E, g, G) した場合，値が正ならば頭に空白文字を付加し，正負が同じ桁数になるようにする。このフラグを指定しない場合は，負の値に限り符合がつく。ただし空白フラグと符合フラグを同時に指定すると，空白フラグは無視される
   - \# 表記フラグ
     数値を変換する場合，どのような方法で変換したかがわかるような形で処理する。
     - o 変換の場合，先頭が “0” (8 進表記文字) になるように精度を増す
     - x 変換の場合，結果が 0 以外ならば “0x” (16 進接頭語) を先頭に付加する
     - X 変換の場合，結果が 0 以外ならば “0X” (16 進接頭語) を先頭に付加する

· e, E, f, G 変換の場合，つねに小数点を付加する (通常は必要なければ省かれる)

· g, G 変換の場合，後続の余分な 0 を取り除かない (通常は取り除く)

· それ以外は未定義

● 0　0 フラグ

結果を 0 詰めする

· d, i, o, u, x, X, e, E, f, g, G 変換の場合，符合や進法表記を除くフィールドの先頭部分を "0" で埋める。ただし左詰めフラグと 0 フラグを同時に指定すると，0 フラグは無視される。また，d, i, o, u, x, X 変換の場合，精度を指定すると 0 フラグは無視される

· それ以外は未定義

2. 最小フィールド幅には，最小のフィールド幅を指定する 10 進整数文字列または "*"(アスタリスク) を指定する。変換の結果が指定した最小フィールド幅よりも小さければ，文字列は最小フィールド幅の間で右詰め/左詰めされる。ただし，最小フィールド幅を指定しなかったり，変換の結果が最小フィールド幅よりも長い場合は何も行わない。もし変換結果の長さを制限したければ，精度を利用すること。

最小フィールド幅に "*" を指定した場合，最小フィールド幅の値は int 型引数から取り出される。このとき，もし値が負であれば，"-" フラグ (符合フラグ) が指定されたとみなし，値が 0 であれば 0 フラグが指定されたとみなす。

3. 精度には "."(ピリオド) から始まり，精度を指定する 10 進整数文字列かまたは "*"(アスタリスク) を指定する。精度に "*" を指定した場合，精度の値は int 型引数から取り出される。このとき値が負の場合は最小フィールド幅とは異なり，単に無視される。

精度は変換指定子によって意味が異なり，もし精度を指定しなかった場合はそれぞれの変換指定子に固有の既定値が取られる。

· d, i, o, u, x, X 変換の場合，精度は出現すべき最小の桁数を意味する (既定値は 1)

· e, E, f 変換の場合，精度は小数点の後に出現すべき桁数を意味する (既定値は 6)

· g, G 変換の場合，精度は最大有効桁数を意味する (既定値は 6)

· s 変換の場合，精度は文字列から取り出される最大バイト数を意味する (既定値は無限大)

· それ以外は未定義

4. 変換修飾子は変換指定子を修飾するもので，数値データなどのデータ型 (データサイズ) を指定するためのものである。変換修飾子には，次の 3 つから 0 個あるいは 1 個を指定することができる。

● h short int 型引数

引数を short int 型に対するものとして扱う

· d, i, o, u, x, X 変換の場合，引数が short int 型であることを示す

· n 変換の場合，引数が short int *型であること示す

· その他は未定義

- l double 型引数と long int 型引数

  引数を double 型あるいは long int 型に対するものとして扱う

  · d, i, o, u, x, X 変換の場合，引数が long int 型であることを示す

  · e, E, f, g, G 変換の場合，引数 double 型であることを示す

  · n 変換の場合，引数が long int *型であること示す

  · その他は未定義

- L long double 型引数と long long int 型引数

  引数を long double 型あるいは long long int 型に対するものとして扱う

  · d, i, o, u, x, X 変換の場合，引数が long long int 型であることを示す

  · e, E, f, g, G 変換の場合，引数 long double 型であることを示す

  · n 変換の場合，引数が long long int *型であること示す

  · その他は未定義

5. **変換指定子**は実際にどういう変換を行うかを指定するもので，必ず1個は指定しなくてはならない。変換指定子には次のようなものがある。

- d, i　整数型引数を符号つき10進数の文字列に変換する
- o　整数型引数を符号なし8進数の文字列に変換する
- x　整数型引数を符号なし16進数の文字列（アルファベット部分は abcdef）に変換する
- X　整数型引数を符号なし16進数の文字列（アルファベット部分は ABCDEF）に変換する
- f　実数型引数を符号つき10進浮動小数の文字列に変換する
- e, E　実数型引数を符号つきで，正規化された指数形式の10進浮動小数文字列に変換する。指数部の表現は e(あるいは E) に符号1文字，そして最低2桁の指数を伴う
- g, G　実数型引数を符号つきで，f あるいは e, E と同じ形式に変換する。しかし小数点は必要がなければ出力されない点，小数点以下の末尾の連続した0が省かれる点，実際の桁数が最大有効桁数より多い場合は最後に丸められた1桁が付加される点が異なる。通常は f 形式が選択されるが，指数が−5以下か精度以上の場合は e, E(g ならば e, G なら E) 形式に切り替わる
- c　int 型引数を unsigned char 型に型変換し，それを文字として出力する

- s　　char *型引数を文字列として出力する
- p　　void *型引数をポインタを表現する形式に変換する
- n　　整数型へのポインタ引数が指す領域に，ここまでに出力したバイト数を格納する
- %　　%文字そのものを出力する

戻り値——正常に出力できた場合は出力した文字数を返し，失敗した場合は EOF を返して，ストリームのエラー指示子を設定する。失敗した場合は変数 errno にその原因を示すエラーコードを設定する。

- EBADF　　ファイルストリームに関連するファイルハンドルがおかしい

注　意——*format*で指定したフォーマット文字のなかで使用される変換指示と，実際に fprintf 関数に渡した引数の数やデータ型が異なっていたり，ずれていたりした場合，その動作は未定義である。たとえば，%c の変換指示は int 型の引数をとるが，これに char 型のデータを渡した場合，どのような値が表示されるかについては未定義である。

互換性——*LIBC*はＣロケールしかサポートしていない。

規　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——eprintf，printf，sprintf

fputc

用　途――ファイルストリームにバイトデータを出力する。

書　式――#include <stdio.h>

```
int fputc (int c, FILE *stream);
```

解　説――fputc 関数は，*stream*で指定されたファイルストリームに対して文字 *c*をunsigned char 型に変換したデータとして出力する。これに応じてストリームのファイルポインタは1バイト分進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形で行われる。

戻り値――正常に出力できた場合は *c*を返し，何らかのエラーによって失敗した場合は，ストリームのエラー指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

規　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――ferror，fopen，putc，puts

fputs

用　途——ファイルストリームに文字列を出力する。

書　式——#include <stdio.h>

```
int fputs (const char *s, FILE *stream);
```

解　説——fputs 関数は，*stream* で指定されたファイルストリームに対して *s* が指す文字列を出力する。文字列は null 文字までとし，最後の null 文字は出力しない。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形で行われる。

戻り値——正常に出力できた場合は負ではない値を返し，失敗した場合は EOF を返してストリームのエラー指示子を設定する。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——ferror，fopen，putc，puts

fread

用　途――ファイルストリームからデータ列を取り込む。

書　式――

```
#include <stdio.h>
int fread (void *ptr, size_t size, size_t n, FILE *stream);
```

解　説――fread 関数は，*stream* で指定したファイルストリームから1項目が *size* バイトのデータ列を *n* 項目分取り込み，結果を *ptr* が指す領域に格納する。それに応じてストリームのファイルポインタは，取り込んだバイト数分だけ進む。

戻り値――実際に取り込んだ項目数を返す。取り込み中にファイルの終端に到達した場合はストリームの終端指示子を設定し，何らかのエラーによって取り込みに失敗した場合はストリームのエラー指示子を設定する。

項目のサイズ *size* あるいは項目数 *n* に0が設定された場合，fread 関数は0を返し，実際には何も行わない。

注　意――ファイルの途中で終端に達したり，エラーが発生した場合，ファイルポインタがどれだけ進むかは未定義である。また項目の途中で中断した場合には，その項目のデータは保証されない。

規　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――feof，ferror，fopen，getc，gets

free

用　　途——メモリブロックを解放する。

書　　式——#include <stdlib.h>

```
void free (void *ptr);
```

解　　説——free 関数は *ptr* が指すメモリ領域を解放し，ヒープ領域に返却する。

戻 り 値——なし。

注　　意——*LIBC* では free 関数でメモリ領域をヒープ領域に返却しても，いったん拡張されたヒープ領域のサイズは小さくはならない。したがって，現在実行中のプロセスが占めるトータルのメモリサイズは変化しない。

なお，*ptr* に malloc 関数，calloc 関数や realloc 関数で確保したメモリ領域へのポインタ以外を指定した場合の動作は未定である。

規　　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——brk，calloc，malloc，rbrk，realloc，sbrk

変　　更——従来では *ptr* に NULL ポインタが指定された場合エラーとなっていたが，現在の *LIBC* では *ANSI C* の規定通り何もしないようにした。

freopen

用　途——ファイルストリームを再オープンする。

書　式——

```
#include <stdio.h>
FILE *freopen (const char *path, const char *mode, FILE *stream);
```

解　説——freopen 関数は，まず *stream* で指定されたファイルストリームをクローズする。次に *path* で指定したファイルをオープンし，対応するファイルストリームを新たに *stream* に割り付ける。*mode* にはストリームのモードを指定することができ，次のなかから1つを選択することができる。ただし，選択はファイルのファイルアクセスモードで許される範囲に限られる。

- "r"　読み込み
- "w"　書き込み
- "a"　追加書き込み
- "r+"　更新 (読み込みと書き込み)
- "w+"　更新 (読み込みと書き込み)
- "a+"　更新 (読み込みと追加書き込み)

ファイルストリームが "w" や "w+" を指定して書き込み可能な形でオープンした場合，すでに *path* で指定したファイルが存在するならばその内容は破棄され，そのサイズは0バイトに切り詰められる。

また，モードを表す文字列の最後に次の文字を指定することで，ストリームのモードをテキスト/バイナリのどちらかにすることができる。省略時はテキストモード，あるいは fmode 関数で指定したモードに合わせられる。

- "b"　バイナリモード
- "t"　テキストモード (CRLF → LF 変換)

"r+b" → バイナリモードでの更新

戻り値——正常にオープンできた場合はストリームへのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EISDIR　ディレクトリである
- ENOSPC　ディスクがいっぱいである
- EROFS　リードオンリーファイルシステムである
- EMFILE　これ以上ファイルストリームを割り当てられない

- EINVAL　*mode*で不正なモードを指定した
- ENOMEM　メモリが足りなくなった
- EACCES　*mode*で指定したモードは許されない

規　格——*ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——fclose, fdopen, fopen, open

F

frexp

用　途——浮動小数点値を仮数部と指数部に分ける。

書　式——

```
#include <math.h>
double frexp (double x, int *exp);
```

解　説——frexp 関数は，xを正規化して小数部と2の累乗を表す整数の指数部に分解し，整数の指数は*exp*が指す領域に格納する。仮数部には，0もしくは$^1/_2 < x \leq 1$の範囲内の数が返される。

戻り値——正しく値が求められた場合は仮数部の値を返す。もしxが非数 (NaN) または無限大だった場合は，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　xの値が非数，または無限大

注　意——つねにソフトウェアエミュレートする。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——ldexp，modf

fscanf

用　途──指定したファイルストリームからフォーマット入力を行う。

書　式──

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format, ...);
```

解　説──fscanf 関数は *format* で指定した入力フォーマット文字列にしたがって，必要なデータ列を *stream* が指すファイルストリームから入力し，その結果を引数の指す領域に格納する。

format に指定するフォーマット文字は通常の文字列であるが，“%” で表される変換指定文字に続く一連の変換方法の書式指定があれば，その指示にしたがって文字列がデータに変換される。これらの変換（たとえば小数点の文字など）は，現在のロケールの LC_NUMERIC カテゴリに左右される。

フォーマット文字に含まれるこれらの変換指定文字以外の部分は，次のように処理される。

1. 空白文字

ストリームからデータを入力し，そのデータが空白文字であるかぎり，それらをすべて読み捨てる

2. 普通の文字

ストリームからデータを入力し，そのデータが指定した文字と一致すればよし，一致しなければそこで fscanf 関数の処理を中断する

通常，変換の書式指定は次のような形式である。

%[代入抑制文字 (*)][最大フィールド幅][変換修飾子][変換指定子]

1. **代入抑制文字**として “*”(アスタリスク) を指定すると，変換は行われるが，実際に値が代入されることはない。特定の入力を無視したい場合に使用する
2. **最大フィールド幅**には，最大のフィールド幅を指定する 10 進整数文字列を指定する。入力されたデータは最大フィールド幅までの範囲で変換される
3. **変換修飾子**は変換指定子を修飾するもので，数値データなどのデータ型（データサイズ）を指定するためのものである。変換修飾子には，次の 3 つから 0 個あるいは 1 個を指定することができる

● h　short int 型引数

引数を short int 型に対するものとして扱う

・d，i，n 変換の場合，引数が short int *型であることを示す

- o, u, x, X 変換の場合，引数が unsigned short int *型であることを示す
- その他は未定義

- l　double 型引数と long int 型引数
  引数を double 型あるいは long int 型に対するものとして扱う
  - d, i, n 変換の場合，引数が long int *型であることを示す
  - o, u, x, X 変換の場合，引数が unsigned long int *型であることを示す
  - e, E, f, g, G 変換の場合，引数 double *型であることを示す
  - その他は未定義

- L　long double 型引数と long long int 型引数
  引数を long double 型あるいは long long int 型に対するものとして扱う
  - d, i, n 変換の場合，引数が long long int *型であることを示す
  - o, u, x, X 変換の場合，引数が unsigned long long int *型であることを示す
  - e, E, f, g, G 変換の場合，引数 long double *型であることを示す
  - その他は未定義

4. **変換指定子**は実際にどういう変換を行うかを指定するもので，必ず1個は指定しなくてはならない。変換指定子には次のようなものがある

- d　符合つき10進整数を整数型に変換し，格納する。符合つき10進整数には先頭の符合は含むが，接尾語のu, U, l, Lは含まない。strtol 関数に基数として10を与えた場合と同じである
- i　符合つき8, 10, 16進整数を整数型に変換し，格納する。符合つき8, 10, 16進整数には先頭の符合を含むが，接尾語のu, U, l, Lは含まない。strtol 関数に基数として0を与えた場合と同じで，数値の基数は文字列の特徴から判別される。たとえば，接頭語として“0”があれば8進数，“0x”や“0X”があれば16進数である
- o　符合つき8進整数を整数型に変換し，格納する。符合つき8進整数には先頭の符合は含むが，接尾語のu, U, l, Lは含まない。strtol 関数に基数として8を与えた場合と同じである
- u　符合なし10進整数を整数型に変換し，格納する。符合なし10進整数には先頭の正(+)符合は含むが，負(-)符合や接尾語のu, U, l, Lは含まない。strtoul 関数に基数として10を与えた場合と同じである

- x,X　符合なし16進整数を整数型に変換し，格納する。符合なし16進整数には先頭の正 (+) 符合および接頭語の "0x" や "0X"(なくてもよい) を含むが，負 (-) 符合や接尾語の u, U, l, L は含まない。変換指定子の x, X はどちらも同じであり，大文字子文字の関係はない。strtoul 関数に基数として16を与えた場合と同じである
- f　符合つき10進浮動小数を実数型に変換し，格納する。符合つき10進浮動小数は fprintf 関数の変換指定子 e, E, f, g, G のいずれの形式でもよく，strtod 関数による変換と同じである。
- s　空白以外の文字列と一致し，その文字列を格納する
- [文字の並び]　"[" と "]" によって囲まれた文字の集合に「含まれる」文字群によって構成される文字列と一致し，その文字列を格納する。集合に "]" を指定したい場合は "[" の直後に指定すること。その場合，最初の "]" は集合の終わりとはみなされない
- [^文字の並び]　"[" と "]" によって囲まれた文字の集合に「含まれない」文字群によって構成される文字列と一致し，その文字列を格納する。集合に "]" を指定したい場合は "^" の直後に指定すること。その場合，最初の "]" は集合の終わりとはみなされない
- c　任意の1文字に一致し，その文字を int 型に変換して格納する
- p　ポインタを表す文字列 (fprintf 関数の p 変換を参照) を void *型に変換して格納する
- n　ここまでに読み込まれたバイト数を格納する。入力は行われない
- %　%文字と一致する。通常の文字処理と同じ

戻り値——正常に入力できた場合は実際に入力できた項目数を返し，失敗した場合は EOF を返して，ストリームの**エラー指示子**を設定する。失敗した場合は変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——引数に指定するポインタは，結果を格納するための十分な大きさの領域を指していなければならない。

また，*format* で指定したフォーマット文字のなかで使用される変換指示と，実際に fscanf 関数に渡した引数の数やデータ型が異なっていたり，ずれていたりした場合，その動作は未定義である。

互 換 性――*LIBC*はCロケールしかサポートしていない。

規　　格――***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――scanf，sscanf

fseek

用　途――ファイルストリームのファイルポインタの位置を再設定する。

書　式――

```
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
```


解　説――fseek 関数は，*stream*で指定したファイルストリームのファイルポインタの位置を再設定する。新しい位置は*whence*で指定された開始位置に，*offset*バイト分のオフセットを加算したものとなる。*whence*は次のマクロ値をとり，それぞれ開始位置を次のように定義する。

- SEEK_SET　ファイルの先頭から計算する
- SEEK_CUR　現在のファイルポインタの位置から計算する
- SEEK_END　ファイルの終端から計算する

fseek 関数を実行すると，バッファに残っている未書き込みデータはフラッシュされ，未読み込みデータは捨てられる。また，ストリームに設定されていた終端指示子はクリアされる。

fseek 関数はファイルの先頭を越えることはできないが，終端を越えてファイルサイズを拡張することはできる。この場合，拡張された部分はすべて 0 で埋められる。

戻り値――正常にシークできた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである
- ESPIPE　ファイルストリームに関連するファイルはシークできない
- EINVAL　不正なオフセットを指定した

互換性――ファイルストリームがテキストモードでオープンされている場合，ftell 関数が正しい値を返す保証はない。したがって，その値を用いてシークしても望む位置にシークできるかどうかは未定義である。

規　格――***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目――fopen，ftell，rewind

fsetpos

用　　途――ファイルストリームのファイルポインタの位置を元に戻す。

書　　式――#include <stdio.h>

```
int fsetpos (FILE *stream, const fpos_t *pos);
```

解　　説――fsetpos 関数は，*stream* で指定したファイルストリームのファイルポインタの位置を *pos* が指す領域から取り出して再設定する。

fsetpos 関数を実行すると，バッファに残っている未書き込みデータはフラッシュされ，未読み込みデータは捨てられる。また，ストリームに設定されていた終端指示子がクリアされる。

戻 り 値――正常に位置を再設定できた場合は0を返し，失敗した場合は−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである
- ESPIPE　ファイルストリームに関連するファイルはシークできない
- EINVAL　不正なオフセットを指定した

注　　意――*pos* の指す領域は，あらかじめ fgetpos 関数によって初期化されている必要がある。

互 換 性――ファイルストリームがテキストモードでオープンされている場合，fgetpos 関数が正しい値を返すという保証はない。したがって，その値を用いてファイルポインタの位置を再設定しようとしても，望む位置に設定されるかどうかは未定義である。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――fgetpos，fopen，fseek，rewind

ftell

用　途——ファイルストリームのファイルポインタの位置を取得する。

書　式——
```
#include <stdio.h>
long ftell (FILE *stream);
```

解　説——ftell 関数は，*stream* で指定されたファイルストリームのファイルポインタの位置を返す。

戻り値——正常にファイルポインタの位置を取得できた場合にはその位置 (ファイルの先頭からのオフセット) を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである
- ESPIPE　ファイルストリームに関連するファイルはシークできない

互換性——ファイルストリームがテキストモードでオープンされている場合，内部で CRLF→LF 変換が行われるため，ftell 関数が正しい値を返すという保証はない。ftell 関数がテキストモードでも正しい値を返すのは，バッファにデータが残っていないとき，すなわちオープン/フラッシュ/シークされた直後だけである。

規　格——*ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——fopen, fseek

ftime

用　　途――現在の時刻を取得する。

書　　式――#include <sys/timeb.h>

```
void ftime (struct timeb *timeptr);
```

解　　説――ftime関数は現在の地域時間を求め，結果を*timeptr*が指す領域に格納する。timeb構造体は次のとおり。

```
struct timeb {
    time_t time;                /* 1970年1月1日から現在までの経過秒数 */
    unsigned short millitim;    /* 経過秒数のミリ秒レベルの値 */
    short timezone;             /* タイムゾーン（グリニッジから西方向へ数える）*/
    short dstflag;              /* サマータイムフラグ */
};
```

戻 り 値――なし。

注　　意――*LIBC*では，timeb構造体のmillitimはつねに0が設定される。

規　　格――***Project LIBC Group***，***4.3BSD***，***MS-C 7.0***

関連項目――ctime，localtime，time，tzset

ftruncate

用　途——ファイルの長さを変更する。

書　式——

```
#include <unistd.h>
int ftruncate (int fildes, off_t len);
```

解　説——ftruncate 関数は，*fildes* が指すファイルの長さを *len* バイトに変更する。もし *len* が現在のファイル長よりも短い場合，その間のデータはすべて捨てられ，反対に長い場合はすべて 0 のデータで埋められる。

ただし，ftruncate 関数はキャラクタデバイスに対しては動作しない。また，ファイルは書き込み可能であることが条件となる。

戻り値——正常に変更できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- EROFS　リードオンリーファイルシステムである

規　格——*AES/OS*

関連項目——chsize，creat，open，truncate

fwrite

用　　途――ファイルストリームにデータ列を書き込む。

書　　式――

```
#include <stdio.h>
int fwrite (void *ptr, size_t size, size_t n, FILE *stream);
```

解　　説――fwrite 関数は *stream* で指定したファイルストリームに対して，1項目が *size* バイトのデータ列を *ptr* が指す領域から取り出して，*n* 項目分書き込む。それに応じてストリームのファイルポインタは，書き込んだバイト数分だけ進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形で行われる。

戻り値――実際に書き込んだ項目数を返す。もし，書き込み中に何らかのエラーによって書き込みに失敗した場合，ストリームのエラー指示子を設定する。

項目のサイズ *size* あるいは項目数 *n* に0が設定された場合は，fwrite 関数は0を返し，実際には何も行わない。

注　　意――ファイルの途中でエラーが発生した場合に，ファイルポインタがどれだけ進むか，すなわち正確に何バイトが書き込まれるかは未定義である。また項目の途中で中断した場合には，その項目のデータは保証されない。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――feof，ferror，fopen，putc，puts

_getdriveno

用　途――ファイル名から論理ドライブ番号を求める。

書　式――#include <sys/xstdlib.h>

```
int _getdriveno (const char *path)
```

解　説――_getdriveno 関数は *path* で指定したファイル名をフルパスに展開し，求められたパスの論理ドライブ番号を返す。

戻り値――正常にドライブ番号が求められた場合は**論理ドライブ番号**を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコード設定する。論理ドライブ番号は A:ドライブを 0 とする。

- ENOENT　不正なドライブを指定した

注　意――_getdriveno 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格――*Project LIBC Group*

関連項目――_dos_getassign，_dos_nameck，_fullentry，_fullpath

_getleaps

用　　途——指定した年までの閏年の回数を調べる。

書　　式——

```
#include <sys/xtime.h>
int _getleaps (int year);
```

解　　説——_getleaps 関数は，1970 年 1 月 1 日 (エポックタイム) から *year* で指定した年の前年までに何回閏年があったかを求める。ここで，閏年とは西暦年が 4 で割り切れる年のことである。ただし閏年でも 100 で割り切れ，400 で割り切れない年は閏年ではない。

戻 り 値——閏年があった回数を返す。

注　　意——_getleaps 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格——*Project LIBC Group*

関連項目——_isleap

gcvt

用　途——浮動小数値をＧフォーマット形式の文字列に変換する。

書　式——#include <stdlib.h>

```
char *gcvt (double value, int prec, char *buff);
```

解　説——gcvt 関数は *value* で指定した浮動小数を G フォーマットの文字列に変換し，その結果を *buff* が指す領域に格納する。変換は，有効桁数が *prec* 桁になるように実行される。*buff* には，変換の結果として小数点や指数表示が格納されることがあるので，*prec* 桁以上の十分な余裕をもつ必要がある。

gcvt 関数は，たとえば次のような結果を返す。

```
gcvt (31.415900, 3) str = '31.4'
gcvt ( 3.141590, 3) str = '3.14'
gcvt ( 0.314159, 3) str = '0.314'
gcvt ( 0.031416, 3) str = '3.14E-002'
gcvt ( 0.003142, 3) str = '3.14E-003'
```

戻り値——*buff* を返す。

注　意——数値は，*prec*+1 桁目で四捨五入によって丸められる。

buff は，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　格——***Project LIBC Group***，***XC***，***MS-C 7.0***

関連項目——ecvt，fcvt

getc

用　　途――ファイルストリームから１バイトを取り出す。

書　　式――

```
#include <stdio.h>
int getc (FILE *stream);
```

解　　説――getc関数は，*stream*で指定されたファイルストリームからunsigned char型のデータとして１バイトを取り出し，int型に変換して返す。ストリームのファイルポインタは，これに応じて１バイト分進む。getc関数は，ungetc関数で押し戻されたデータを正しく扱うことができないこと，および更新モードで正しく動作しないことを除けばfgetc関数と同じである。

戻り値――正常に取り出せた場合は，そのデータを返す。もしストリーム操作中に何らかのエラーが発生した場合はストリームの**エラー指示子**を設定してEOFを返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定してEOFを返す。失敗した場合は，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　　意――getc関数は，デフォルトではマクロとして定義されるが，__NO_STDIO_INLINE__が定義された場合は実体をもつ関数となる。

getc関数がマクロとして展開される場合，引数が副作用を伴わないように注意すること。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――fgetc

getch

用　途——コンソールから直接1文字を入力する(エコーなし)。

書　式——#include <conio.h>

```
int getch (void);
```

解　説——getch関数はコンソールから1文字を入力し，その文字コードを返す。ただし入力された文字は，コンソールにエコーバックされない。getch関数はつねにコンソールから直接データを読み取るので，標準入力の状態に左右されず，また stdio ライブラリによるバッファリングの作用も受けない。

getch関数は1文字を入力するまで待つが，すでにキーボードバッファにデータがある場合や ungetch関数によって押し戻されたデータがある場合は，それを取得する。

戻り値——入力された文字の文字コードを返す。エラーはない。

注　意——構造上，0x00の文字コードは取得することができない。

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——getc，getchar，getche

getchar

用　　途――標準入力ファイルストリームから１バイトを取り出す。

書　　式――#include <stdio.h>

```
int getchar (void);
```

解　　説――getchar 関数は，標準入力に割り当てられたファイルストリームから unsigned char 型のデータとして１バイトを取り出し，それを int 型に変換して返す。ストリームのファイルポインタは，これに応じて１バイト分進む。

getchar 関数は ungetc 関数で押し戻されたデータを正しく扱うことができないこと，および更新モードで正しく動作しないことを除けば，fgetc 関数に stdin を指定した場合と同じである。

戻り値――正常に取り出せた場合は，そのデータを返す。もしストリーム操作中に何らかのエラーが発生した場合は標準入力ストリームのエラー指示子を設定して EOF を返し，ファイルの終端に到達した場合はストリームの終端指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　　意――getchar 関数はデフォルトではマクロとして定義されるが，__NO_STDIO_INLINE__が定義された場合は実体をもつ関数となる。

getchar 関数がマクロとして展開される場合，引数は副作用を伴わないように注意すること。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――fgetc，getc

getche

用　　途——コンソールから直接 1 文字を入力する (エコーあり)。

書　　式——`#include <conio.h>`

```
int getche (void);
```

解　　説——getche 関数はコンソールから 1 文字を入力し，その文字コードを返す。getch 関数とは異なり，入力された文字はコンソールにエコーバックされる。getche 関数はつねにコンソールから直接データを読み取るので，標準入力の状態に左右されることも stdio ライブラリによるバッファリングの作用も受けない。

getche 関数は 1 文字を入力するまで待つが，すでにキーボードバッファにデータがある場合や ungetch 関数によって押し戻されたデータがある場合は，それを取得する。

戻 り 値——入力された文字の文字コードを返す。エラーはない。

注　　意——構造上，0x00 の文字コードは取得することができない。

規　　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——getc，getch，getchar

getcwd

用　　途──カレントワーキングディレクトリを取得する。

書　　式──#include <unistd.h>

```
char *getcwd (char *buff, int size);
```

解　　説──getcwd関数は，カレントドライブのカレントワーキングディレクトリのパス名を取得し，*buff*が指す*size*バイトの領域にコピーする。パス名は，すべて先頭にドライブ名を付加した絶対パス名の形で返される。

"A:/foo/bar/sample/directory"

戻 り 値──正常に取得できた場合には*buff*を返し，失敗した場合にはNULLを返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　*size*が0または負の値である
- ERANGE　パス名が*size*バイトより長い

規　　格──***Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目──chdir，chdrive，getdcwd

変　　更──従来では*buff*にNULLポインタが指定された場合エラーとなっていたが，現在の*LIBC*ではgetcwd関数が結果を格納する領域を自動的に割り当てるようにした。

getdcwd

用　途——指定ドライブのカレントワーキングディレクトリを取得する。

書　式——#include <unistd.h>

```
char *getdcwd (int drive, char *buff, int size);
```

解　説——getdcwd 関数は，*drive* で指定したドライブのカレントワーキングディレクトリのパス名を取得し，*buff* が指す *size* バイトの領域にコピーする。パス名は，ドライブ名なしの絶対パス名で返される。*drive* は 0 がカレントドライブ，以降，1 が A:ドライブ，2 が B:ドライブの順番で指定する。

"/foo/bar/baz"

戻り値——正常に取得できた場合には *buff* を返し，失敗した場合には NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　*size* が 0 または負の値である
- ERANGE　パス名が *size* バイトより長い

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——chdir，chdrive，getcwd

変　更——従来では *buff* に NULL ポインタが指定された場合エラーとなっていたが，現在の *LIBC* では getdcwd 関数が結果を格納する領域を自動的に割り当てるようにした。

getdrive

用　　途――カレントドライブを取得する。

書　　式――

```
#include <unistd.h>
int getdrive (void);
```

解　　説――getdrive関数はカレントドライブを取得する。

戻り値――getdrive関数はカレントドライブを返す。エラーはない。ドライブ番号はA:ドライブを1とし，以降，2がB:ドライブ，3がC:ドライブの順番に割り振られる。

規　　格――*Project LIBC Group*，　*MS-C 7.0*

関連項目――chdir，chdrive，getcwd，getdcwd

getegid

用　途——実効グループ ID を取得する。

書　式——`#include <unistd.h>`

```
gid_t getegid (void);
```

解　説——getegid 関数は，カレントプロセスの実効グループ ID を返す。

戻り値——実効グループ ID を返す。

G

互換性——**Human68k** にはグループ ID の概念がない。そのため getegid 関数では，環境変数 EGID の値を実効グループ ID とする。しかし，失敗した場合は getgid 関数の戻り値を実効グループ ID とする。

規　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——getgid，setgid

getenv

用　途——環境変数の値を取得する。

書　式——

```
#include <stdlib.h>
char *getenv (const char *name);
```

解　説——getenv 関数はプロセスの環境変数テーブルを検索し，*name*で指定された環境変数の値を取得する。

戻り値——正常に取得できた場合は環境変数の値 (文字列) へのポインタを返し，失敗した場合には NULL を返す。

注　意——アプリケーションは取得されたポインタの指す領域を変更してはならない。

検索される環境変数テーブルはプロセス内部にコピーされた環境変数テーブルであり，親プロセスの環境変数テーブルや **Human68k** の環境変数テーブルではない。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——clearenv，environ，putenv

geteuid

用　途——実効ユーザ ID を取得する。

書　式——`#include <unistd.h>`

```
uid_t geteuid (void);
```

解　説——`geteuid` 関数はカレントプロセスの実効ユーザ ID を返す。

戻 り 値——実効ユーザ ID を返す。

互 換 性——**Human68k** にはユーザ ID の概念がない。そのため `geteuid` 関数では，環境変数 `EUID` の値を実効ユーザ ID とする。しかし，失敗した場合は `getuid` 関数の戻り値を実効ユーザ ID とする。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——`getuid`，`setuid`

getgid

用　途――実グループIDを取得する。

書　式――

```
#include <unistd.h>
gid_t getgid (void);
```

解　説――getgid関数はカレントプロセスの実グループIDを返す。

戻り値――実グループIDを返す。

互換性――**Human68k**にはグループIDの概念がない。そのためgetgid関数では，環境変数GIDの値を実グループIDとする。しかし，失敗した場合は0 (root)を実グループIDとする。

規　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――getegid，setgid

getgrent

用　途——グループファイルから1データを取り出す。

書　式——#include <grp.h>

```
struct group *getgrent (void);
```

解　説——getgrent 関数は，呼び出されるごとにグループファイルから1グループデータを取り出して，そのグループデータへのポインタを返す。グループファイルは最初の getgrent 関数か setgrent 関数の呼び出しでオープンされ，endgrent 関数でクローズされるまで連続して利用される。

読み出されたグループデータはフォーマット処理され，getgrent 関数内部の静的領域に格納される。group 構造体で表されるグループデータの構造は次のとおり。

```
struct group {
    char    *gr_name;   /* グループ名 */
    char    *gr_passwd; /* グループパスワード */
    gid_t    gr_gid;    /* グループ ID */
    char   **gr_mem;    /* グループメンバの配列 */
};
```

グループファイルは固定的に“A:/etc/group”が用いられるが，環境変数 SYSROOT によってこれらのファイルのルートディレクトリを変更することもできる。たとえば，SYSROOT に“B:/root”が設定されている場合，グループファイルのパス名は“B:/root/etc/group”となる。

戻り値——正常にデータを取り出せた場合は，グループデータへのポインタを返す。すべてのデータを読み終わったり，グループファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　グループファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意——結果は関数内部の静的領域に格納されるため，getgrent 関数，getgrnam 関数，getgrgid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

互 換 性——グループファイルのファイルフォーマットは，各項目間のセパレータに':'ではなくて';'を用いている以外は**UNIX**と同じである。

規　　格——***Project LIBC Group***，***4.3BSD***，***SYSV***

関連項目——endgrent，getgrgid，getgrnam，setgrent

getgrgid

用　途——グループファイルからグループ ID でデータを検索する。

書　式——#include <grp.h>

```
struct group *getgrgid (gid_t gid);
```

解　説——getgrgid 関数は，グループファイルから *gid* で指定したグループ ID のデータを検索し，そのグループデータへのポインタを返す。グループファイルは検索終了後に自動的にクローズされるので，endgrent 関数を呼び出す必要はない。

group 構造体の構造など，詳細は getgrent 関数を参照のこと。

戻り値——該当するデータを見つけることができた場合はグループデータへのポインタを返し，見つからなかったか，グループファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　グループファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意——結果は関数内部の静的領域に格納されるため，getgrent 関数，getgrnam 関数，getgrgid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——endgrent，getgrent，getgrnam，setgrent

getgrnam

用　途――グループファイルからグループ名でデータを検索する。

書　式――#include <grp.h>

```
struct group *getgrnam (const char *name);
```

解　説――getgrnam 関数はグループファイルから *name* で指定したグループ名のデータを検索し，そのグループデータへのポインタを返す。グループファイルは検索終了後に自動的にクローズされるので，endgrent 関数を呼び出す必要はない。

group 構造体の構造など，詳細は getgrent 関数を参照のこと。

戻り値――該当するデータを見つけることができた場合はグループデータへのポインタを返し，見つからなかったり，グループファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　グループファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意――結果は関数内部の静的領域に格納されるため，getgrent 関数，getgrnam 関数，getgrgid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――endgrent，getgrent，getgrgid，setgrent

getlogin

用　途——ユーザのログイン名を取得する。

書　式——#include <unistd.h>

```
char *getlogin (void);
```

解　説——getlogin 関数はカレントプロセスの制御端末へのログイン名を調べ，そのログイン名へのポインタを返す。

戻り値——正常に取得できた場合はログイン名へのポインタを返し，失敗した場合には NULL を返す。

注　意——結果は関数内部の静的領域に格納されるため，関数を呼び出すたびに内容が上書きされることに注意すること。

互換性——Human68k は制御端末およびユーザ ID に関する概念がない。そのため，getlogin 関数では ksh，bash，fish，minsh などのシェルと同様，環境変数 USER あるいは LOGNAME からログイン名を取得することにしている。なお，これらの環境変数が設定されていない場合は “root” を返す。

規　格——*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——cuserid，getpwnam，getpwuid，getuid

getopt

用　途──引数配列からオプション文字列を解析する。

書　式──

```
#include <stdlib.h>
int getopt (int argc, const char *argv[], const char *optstring);
extern char *optarg;
extern int optind;
extern int opterr;
extern int _getopt_no_reordering;
```

解　説──getopt関数は，コマンドラインを解析するためのものである。getopt関数は*argc*と*argv*とで指定されたコマンドラインの引数配列を*optstring*で指定されたオプション文字列にしたがって走査する。

*optstring*に指定するオプション文字列には，使用することのできるオプション文字の一覧を指定する。たとえば，オプションとして“-a”と“-v”を指定できるようにしたいならば，*optstring*には“av”と指定する。

また，これらのオプション文字の直後に“:”をつけることによって，オプションが引数を取るということを表すことができる。この場合，オプションの引数は，

```
-a␣bar
-abar
```

getopt関数は呼ばれるたびに変数optindが指す位置の次の引数を調べ，もしそれが“-”で始まり，かつ*optstring*に指定されているオプション文字であれば，その文字コードを返す。また同時に，変数optindに現在の位置を記録し，オプションが引数を取るならば変数optargにその引数へのポインタを設定する。なおライブラリは，最初のgetopt関数の呼び出しにさきだって変数optindを0にクリアする。

通常，getopt関数は“-”で始まるオプションを見つけると，*optstring*に指定されたオプション文字列と照合する。しかしこれに含まれない，すなわち不正なオプション文字を発見したり，引数を必要とするのに引数が指定されていないような場合には，標準エラー出力にエラーメッセージを出力し，“?”を返す。もし，このエラーメッセージを出力しないようにするには，変数opterrを0に設定する。デフォルトでは1が設定されている。

getopt関数は，これ以上オプション文字が見つからないと判断するとEOFを返し，変数optindにオプション文字ではない引数のうちで先頭のものの位置を設定する。たとえば，

```
-a -b foo.c bar.c
```

という引数配列があり，これに対して *optstring* に "ab:" を設定した場合，最初の呼び出しで getopt 関数は "a" を返し，2 回目の呼び出しでは "b" を返すのに加えて変数 optarg に "foo.c" へのポインタを返す。そして 3 回目の呼び出しでは，これ以上オプション引数がないため EOF を返し，変数 optind に "bar.c" のインデックスを設定する。

一般に getopt 関数は引数列を前から解析していき，オプションではない引数を発見した段階で処理を中断してしまう (EOF を返す) が，*LIBC* の getopt 関数は引数列全体を走査し，すべてのオプション指定を調べることができる。またその結果，引数配列はオプション指定が先頭に，そうではない引数が後ろになるようにソートされる。もしこうしたソート機能に支障があれば，変数_getopt_no_reordering に 1 を設定する。デフォルトでは 0 が設定される。

この機能を詳しく説明しよう。たとえば，

```
-a -cfoo.c -v foo.c bar.c baz.c -z -t foo.c
```

という引数配列に対して *optstring* に "ac:v:zt:" を指定し，ソートが行われる状態で getopt 関数を実行してみよう。すると，getopt 関数が EOF を返して捜査が終了した段階で，引数列は次のようにソートされ，変数 optind は "bar.c" を指すことになる。

```
-a -cfoo.c -v foo.c -z -t foo.c bar.c baz.c
```

戻り値——有効なオプションを見つけるとそのオプション文字の文字コードを返し，無効なオプションを見つけると "?" を返す。また，すべてのオプションの処理が終わると EOF を返し，"--" というオプションを発見すると，特殊な処理として，そこですべてのオプションが終了したものとして EOF を返す。

注　意——デフォルト状態では引数配列のソートが行われ，その結果，*argv* の中身が書き換えられることがある。

互換性——デフォルトで引数配列のソートが行われるため，これが行われないものと仮定したうえで書かれたプログラムとは互換性を失う。その場合は，変数 _getopt_no_reordering を最初の getopt 関数呼び出しの前に 0 にしておくことが必要となる。

オプションかどうかを判定するのは，引数の先頭に "-" があるかどうかである。**Human68k** や **MS-DOS** でよく使われる "/" は，パス名との区別ができないために *LIBC* では使用していない。

規　格——***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

サンプル——List 1-2 ● getopt 関数の使用例

```
/*
** オプション a, b を解析する
*/

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    int c;

    while ((c = getopt (argc, argv, "ab:")) != EOF) {
        switch (c) {
        case 'a':
            printf ("OPTION A\n");
            break;
        case 'b':
            printf ("OPTION B --- %s\n", optarg);
            break;
        case '?':
        default:
            printf ("UNKNOWN OPTION\n");
            break;
        }
    }
    return 0;
}
```

getpgrp

用　途——プロセスグループ ID を取得する。

書　式——#include <unistd.h>

```
pid_t getpgrp (void);
```

解　説——getpgrp 関数はカレントプロセスの**プロセスグループ ID** を調べ，その値を返す。

戻り値——プロセスグループ ID を返す。

互換性——**Human68k** はマルチタスクではないので，プロセス ID およびプロセスグループ ID の概念がない。そのため getpgrp 関数はつねに 1 (init プロセス) を返す。

規　格——***POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——getpid，getppid，setpgid，setsid

getpid

用　途——プロセスIDを取得する。

書　式——

```
#include <unistd.h>
pid_t getpid (void);
```

解　説——getpid関数はカレントプロセスのプロセスIDを返す。

戻り値——プロセスIDを返す。

互換性——**Human68k**はマルチタスクではないのでプロセスIDの概念がない。そのためgetpid関数では，カレントプロセスのロードアドレス，すなわちメモリ管理ポインタの値をプロセスIDとする。

UNIXでは，プロセスIDは0～30000の範囲内に定められているが，メモリ管理ポインタの値は24ビットなので範囲を越えてしまう。したがって，この定義に忠実にコーディングされているプログラムとは互換性を失う。

規　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——getpgrp，getppid，kill，setsid

getppid

用　途──親プロセス ID を取得する。

書　式──`#include <unistd.h>`

```
pid_t getppid (void);
```

解　説──getppid 関数は，カレントプロセスの親プロセスのプロセス ID を返す。

戻り値──親プロセス ID を返す。

互換性──**Human68k** はマルチタスクではないので，プロセス ID の概念がない。そのため getppid 関数では，親プロセス ID としてカレントプロセスのメモリ管理ポインタ の親メモリ管理ポインタの値を用いる。

UNIX では，プロセス ID は 0 ～ 30000 の範囲内に定められているが，メモリ管理ポインタの値は 24 ビットなので範囲を越えてしまう。したがって，この定義に忠実にコーディングされているプログラムとは互換性を失う。

規　格──***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目──getpgrp，getpid，kill，setsid

getpwent

用　途——パスワードファイルから1データを取り出す。

書　式——#include <pwd.h>

```
#include <pwd.h>
struct passwd *getpwent (void);
```

解　説——getpwent 関数は呼び出されるごとにパスワードファイルから1パスワードデータを取り出して，そのパスワードデータへのポインタを返す。パスワードファイルは最初の getpwent 関数か setpwent 関数の呼び出しでオープンされ，endpwent 関数でクローズされるまで連続して利用される。

読み出されたパスワードデータはフォーマット処理され，getpwent 関数内部の静的領域に格納される。passwd 構造体で表されるパスワードデータの構造は次のとおり。

```
struct passwd {
    char    *pw_name;   /* ユーザ名 */
    char    *pw_passwd; /* パスワード */
    uid_t    pw_uid;    /* ユーザ ID */
    gid_t    pw_gid;    /* グループ ID */
    char    *pw_gecos;  /* GECOS フィールド */
    char    *pw_dir;    /* ホームディレクトリ */
    char    *pw_shell;  /* ログインシェル */
};
```

パスワードファイルは固定的に “A:/etc/passwd” が用いられるが，環境変数 SYSROOT によって，これらのファイルのルートディレクトリを変更することもできる。たとえば SYSROOT に “B:/root” が設定されている場合，パスワードファイルのパス名は “B:/root/etc/passwd” となる。

戻り値——正常にデータを取り出せた場合はパスワードデータへのポインタを返し，すべてのデータを読み終わったか，パスワードファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　パスワードファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意——結果は関数内部の静的領域に格納されるため，getpwent 関数，getpwnam 関数，getpwuid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

互 換 性——パスワードファイルのファイルフォーマットは，各項目間のセパレータが‘:’ではなくて‘;’を用いている以外はUNIXと同じであり，ITA ToolBoxの“login.x”が用いているフォーマットに合わせてある。

規　　格——*Project LIBC Group*，*4.3BSD*，*SYSV*

関連項目——endpwent，getpwnam，getpwuid，setpwent

getpwnam

用　途——パスワードファイルからユーザ名でデータを検索する。

書　式——#include <pwd.h>

```
struct passwd *getpwnam (const char *name);
```

解　説——getpwnam 関数はパスワードファイルから *name* で指定したユーザ名のデータを検索し，そのパスワードデータへのポインタを返す。パスワードファイルは検索終了後に自動的にクローズされるので，endpwent 関数を呼び出す必要はない。

passwd 構造体の構造など，詳細は getpwent 関数を参照のこと。

戻り値——該当するデータを見つけることができた場合はパスワードデータへのポインタを返し，見つからなかったか，パスワードファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　パスワードファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意——結果は関数内部の静的領域に格納されるため，getpwent 関数，getpwnam 関数，getpwuid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——endpwent，getpwent，getpwuid，setpwent

getpwuid

用　途——パスワードファイルからユーザ ID でデータを検索する。

書　式——#include <pwd.h>

```
struct passwd *getpwuid (uid_t uid);
```

解　説——getpwuid 関数はパスワードファイルから *uid* で指定したユーザ ID のデータを検索し，そのパスワードデータへのポインタを返す。パスワードファイルは検索終了後に自動的にクローズされるので，endpwent 関数を呼び出す必要はない。

passwd 構造体の構造など，詳細は getpwent 関数を参照のこと。

戻り値——該当するデータを見つけることができた場合はパスワードデータへのポインタを返し，見つからなかったか，パスワードファイルをオープンできないなど何らかの原因で失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOENT　パスワードファイルが見つからない
- ENOMEM　メモリが足りなくなった

注　意——結果は関数内部の静的領域に格納されるため，getpwent 関数，getpwnam 関数，getpwuid 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——endpwent，getpwent，getpwnam，setpwent

getrlimit

用　　途――システム制限値を取得する。

書　　式――#include <sys/resources.h>

```
int getrlimit (int resource, struct rlimit *rlp);
```

解　　説――getrlimit関数は現在の環境における*resource*で指定したシステム制限値を取得し，結果を*rlp*が指す領域に格納する。*resource*には次の値を指定することができ，その結果はそれぞれ以下のようにして求められる。

- RLIMIT_CPU　CPU消費時間は無制限(RLIM_INFINITY)
- RLIMIT_FSIZE　最大ファイルサイズは**Human68k**が認識する最大のハードディスクに作成できる最大のファイルのサイズ(概算)
- RLIMIT_DATA　ヒープ領域の最大サイズは残りメモリいっぱいまで
- RLIMIT_STACK　スタック領域の最大サイズは最初に確保したサイズまで
- RLIMIT_CORE　コアファイルの最大サイズは環境変数limit_coreに指定されたバイト数まで
- RLIMIT_RSS　メモリ空間の最大サイズは残りメモリいっぱいまで
- RLIMIT_NOFILE　最大オープンファイル数はOPEN_MAXまで

また，rlimit構造体は次のとおり。

```
struct rlimit {
    int rlim_cur; /* 現在の制限値 */
    int rlim_max; /* 設定できる最大の制限値 */
};
```

戻 り 値――正常に取得できた場合は0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　不正な*resource*を指定した

規　　格――***Project LIBC Group***，　***4.3BSD***

関連項目――setrlimit

gets

用　途——標準入力ファイルストリームから文字列を取り出す。

書　式——#include <stdio.h>

```
char *gets (char *s);
```

解　説——gets 関数は**標準入力**に割り当てられた**ファイルストリーム**から文字列を取り出し，結果を *s* が指す領域に格納する。文字列の取り出しはファイルの終端に到達するか，**改行文字**に到達するまで続けられる。結果の文字列には改行文字は含まれず，最後に null 文字が置かれることに注意すること。

戻り値——正しく文字列を取り出すことができた場合には *s* を返す。もしストリーム操作中に何らかのエラーが発生した場合はストリームの**エラー指示子**を設定して NULL を返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定して NULL を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——gets 関数は fgets 関数とは異なり，結果の文字列に改行文字を含まないことに注意すること。また *s* は結果を格納するのに十分な領域を指していなければならない。

規　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——ferror，fread

getuid

用　途――実ユーザID を取得する。

書　式――

```
#include <unistd.h>
uid_t getuid (void);
```

解　説――getuid 関数はカレントプロセスの実ユーザ ID を返す。

戻り値――実ユーザ ID を返す。

互換性――**Human68k** にはユーザ ID の概念がない。そのため getuid 関数では，環境変数 UID の値を実ユーザ ID とする。しかし，失敗した場合は 0 (root) を実ユーザ ID とする。

規　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――geteuid，setgid

getw

用　途——ファイルストリームからワードデータを取り出す。

書　式——#include <stdio.h>

```
int getw (FILE *stream);
```

解　説——getw 関数は，*stream*で指定したファイルストリームからワードデータを取り出す。ワードデータは int 型であり，このサイズは一般にマシンに依存するが，ここでは 4 バイトロングワードとする。なお getw 関数は，ファイルストリームに対してはアラインメント計算を行わない。

戻り値——正しくワードデータを取り出すことができた場合にはその値を返す。もしストリーム操作中に何らかのエラーが発生した場合はストリームのエラー指示子を設定して EOF を返し，ファイルの終端に到達した場合はストリームの終端指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

互換性——*XC*では getw 関数が取り出すのは 2 バイトショートワードだが，*LIBC*では本来の定義どおり，int 型を表現する 4 バイトロングワードを取り出すこととする。したがって，getw 関数を用いて作成されたプログラムとは互換性を失う。

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

gmtime

用　　途――暦時間を協定世界時間(UTC)に変換する。

書　　式――#include <time.h>

```
struct tm *gmtime (const time_t *timep);
```

解　　説――gmtime関数は，*timep*が指す領域に格納された暦時間(エポックタイム)を協定世界時間(UTC)に変換し，その結果をtm構造体に格納してそのアドレスを返す。tm構造体は次のとおり。

```
struct tm {
    int   tm_sec;     /* 秒 (0-61) */
    int   tm_min;     /* 分 (0-59) */
    int   tm_hour;    /* 時 (0-23) */
    int   tm_mday;    /* 日 (1-32) */
    int   tm_mon;     /* 月 (0-11) */
    int   tm_year;    /* 年 (年 - 1900) */
    int   tm_wday;    /* 曜日 (0-7 日曜は0) */
    int   tm_yday;    /* 年内の通算日 (0-365) */
    int   tm_isdst;   /* サマータイム (0-1 実施中なら1) */
    int   tm_gmtoff;  /* グリニッジから西回り */
    char *tm_zone;    /* タイムゾーン名 */
};
```

戻 り 値――正常に変換できた場合はtm構造体へのポインタを返し，失敗した場合にはNULLを返す。

注　　意――結果は関数内部の静的領域に格納されるため，localtime関数，gmtime関数，asctime関数，ctime関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――asctime，ctime，localtime，tzset

IJUMP

用　　途——rts 命令を用いた大域ジャンプを実行する。

書　　式——#include <interrupt.h>

```
void IJUMP (void *addr);
```

解　　説——IJUMP はrts 命令をうまく用いて，*addr*で指定したアドレスに大域ジャンプを行う。これは，割り込みなどで元のベクタにジャンプするために使用する。なおIJUMP が行われる前には，通常の後処理と同じように，関数内部で破壊したレジスタはすべて復帰される。

戻 り 値——なし。IJUMP は決して戻ってこない。

注　　意——元の割り込みベクタへのジャンプなど，限られた用途にだけ使用すること。もし一般的な大域ジャンプを行いたい場合には，setjmp 関数および longjmp 関数を使用しなければならない。

互 換 性——この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された GCC でしか使用することができない。移植性を考慮するならば使用しないこと。

規　　格——*Project LIBC Group*

関連項目——IJUMP_RTE，IRTE，IRTS

IJUMP_RTE

用　　途――割り込みルーチンの宣言とrte命令へのジャンプを実行する。

書　　式――#include <interrupt.h>

```
void IJUMP_RTE (void);
```

解　　説――IJUMP_RTEは割り込み処理関数でのrte命令へのジャンプを行う。

IJUMP_RTEが使用されると，その関数は純粋な割り込み処理関数としてコンパイルされる。つまり，その関数から外部の関数を呼び出さなければその関数で破壊されるレジスタだけが，呼び出す場合はすべてのレジスタが退避/復帰されるようにコンパイルされる。

戻り値――なし。IJUMP_RTEは決して戻ってこない。

互換性――この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された**GCC**でしか使用することができない。移植性を考慮するならば使用しないこと。

規　　格――*Project LIBC Group*

関連項目――IJUMP，IRTE，IRTS

IRTE

用　　途——割り込みルーチンの宣言と rte 命令による復帰を実行する。

書　　式——#include <interrupt.h>
void IRTE (void);

解　　説——IRTE は rte 命令を用いて，割り込みルーチンから復帰する。割り込みルーチンでは，return の代わりに IRTE を使用すること。

IRTE が使用されると，その関数は純粋な**割り込み処理関数**としてコンパイルされる。つまり，その関数から外部の関数を呼び出さない場合はその関数で破壊されるレジスタだけが，呼び出す場合はすべてのレジスタが退避/復帰されるようにコンパイルされる。

戻 り 値——なし。IRTE は決して戻ってこない。

注　　意——IRTE と IRTS は同時には使用しないこと。また，割り込み処理関数以外では使用しないこと。使用した場合，その動作は不定である。

互 換 性——この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された **GCC** でしか使用することができない。移植性を考慮するならば使用しないこと。

規　　格——***Project LIBC Group***

関連項目——IJUMP，IJUMP_RTE，IRTS

IRTS

用　途——全レジスタを保存する関数を宣言する。

書　式——#include <interrupt.h>
void IRTS (void);

解　説——IRTSはrts命令を関数から復帰する。レジスタ退避型関数では，returnの代わりにIRTSを使用すること。

IRTSが使用されると，その関数はレジスタ退避型関数としてコンパイルされる。つまり関数内部で使用される/されないに関わらず，すべてのレジスタが保存されるようにコンパイルされる。

戻り値——なし。IRTSは決して戻ってこない。

注　意——IRTSとIRTEは同時には使用しないこと。

互換性——この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された**GCC**でしか使用することができない。移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——IJUMP，IJUMP_RTE，IRTE

_is68881

用　途——数値演算コプロセッサの種類を調べる。

書　式——

```
#include <sys/xmath.h>
int _is68881 (void);
```

解　説——_is68881 関数は数値演算コプロセッサが実装されているかどうかを調べ，実装されている場合はその種類を返す。

戻り値——数値演算コプロセッサが実装されていた場合はその種類を返し，実装されていない場合は−1 を返す。

- MC68882　　0 を返す
- MC68881　　1 を返す

注　意——数値演算コプロセッサの種類を区別するため，内部の ROM OFFSET $01 を読み出して判定している。この部分は未定義であり，将来変更される可能性がある。

_is68881 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——*Project LIBC Group*

_isleap

用　　途――閏年補正の必要性について調べる。

書　　式――#include <sys/xtime.h>

```
int _isleap (int month, int year);
```

解　　説――_isleap関数は，*year*で指定した年の*month*月が閏年補正をすべきかどうかについて調べる。閏年補正を行わなければならないのは閏年の2月だけであり，閏年とは西暦年が4で割り切れる年のことである。ただし閏年でも100で割り切れ，400で割り切れない年は閏年ではない。

戻 り 値――閏年補正をしなくてはならなければ1を，しなくてよいならば0を返す。

注　　意――単純に*year*の値を評価したいだけならば，*month*には2を指定するとよい。

_isleap関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格――***Project LIBC Group***

index

用　　途——文字列中から指定文字を検索する。

書　　式——#include <string.h>

```
char *index (const char *string, int character);
```

解　　説——index 関数は，*string* が指す文字列中から *character* で指定された文字を検索する。*string* 末尾の null 文字も検索の対象となりうる。*character* は，内部で int 型から char 型に変換される。

戻 り 値——*string* 中で最初に現れた *character* へのポインタを返し，見つからない場合は NULL を返す。

注　　意——1 文字は 1 バイトとなる。

規　　格——***4.3BSD***

関連項目——memchr, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strtok

intlevel

用　　途——CPUステータスレジスタの割り込みマスクを設定する。

書　　式——#include <interrupt.h>

```
void intlevel(level);
```

解　　説——intlevel関数は，CPUステータスレジスタの割り込みマスクビットである(I_2, I_1, I_0)に*level*の値を設定する。ユーザモードからの設定が可能である。

戻 り 値——変更前の割り込みマスク(I_2, I_1, I_0)の値を返す。

規　　格——***Project LIBC Group***

サンプル——**List 1-3 ● スプリアス割り込みを起こさないMFPレジスタのアクセス**

```
#include <stdio.h>
#include <interrupt.h>
#include <sys/dos.h>

#define MFP_INT_MASK(mask)                                  \
    ({                                                      \
        register unsigned short ret;                        \
        asm volatile (                                      \
                     "lea.l $e88013,a0\n\t"                 \
                     "movep.w 0(a0),%0\n"                   \
                     "move.w %c1,d1\n"                      \
                     "movep.w d1,0(a0)\n"                   \
                     : "=d" ((unsigned short) (ret))        \
                     : "d"  ((unsigned short) (mask))       \
                     : "d1","a0");                          \
                     ret;                                   \
    })

void main (void)
{
    int lv, ssp;
    unsigned short mfpmask;

    /* 割り込みをマスクする */
    lv = intlevel (7);

    /* スパーバイザモードにする */
    ssp = _dos_super (0);

    /* MFP MASK を設定 */
    mfpmask = MFP_INT_MASK (0);
    printf ("MFP MASK = %X\n", mfpmask);

    /* MFP MASK を元に戻す */
    mfpmask = MFP_INT_MASK (mfpmask);
    printf ("MFP MASK = %X\n", mfpmask);

```

```
    /* スーパーバイザモードを元に戻す */
    if(ssp > 0)
        _dos_super (ssp);

    /* 割り込みレベルを元に戻す */
    intlevel (lv);
}
```

isalnum

用　　途――英数字かどうかを調べる。

書　　式――`#include <ctype.h>`
`int isalnum (int c);`

解　　説――isalnum関数は，*c*で指定した文字コードを現在のロケールのLC_CTYPEカテゴリに照らし合わせ，**英数字**であるかどうかを調べる。

戻り値――*c*が英数字であれば0以外の値を返し，異なれば0を返す。

注　　意――*c*はint型の引数だが，その値はunsigned char型で表現できる範囲の値かあるいはEOFと同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isalnum関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isalnum関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はCロケールしかサポートしていない。

規　　格――***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper，isxdigit

isalpha

用　途——アルファベットかどうかを調べる。

書　式——#include <ctype.h>

```
#include <ctype.h>
int isalpha (int c);
```

解　説——isalpha 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，英字 (アルファベット) であるかどうかを調べる。

戻り値——*c*が英字 (アルファベット) であれば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isalpha 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isalpha 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——isalnum, isascii, iscntrl, isdigit, isgraph, isiso, islower, isodigit, isprint, ispunct, isspace, isupper, isxdigit

isascii

用　　途——7ビットASCII文字かどうかを調べる。

書　　式——#include <ctype.h>

```
int isascii (int c);
```

解　　説——isascii関数は，*c*で指定した文字コードが7ビットASCII文字(0x00 ～ 0x7F)であるかどうかを調べる。

戻 り 値——*c*が7ビットASCII文字であれば0以外の値を返し，異なれば0を返す。

注　　意——*c*はint型の引数だが，その値はunsigned char型で表現できる範囲の値か，あるいはEOFと同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isascii関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isascii関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

isascii関数はロケールとは関係はない。

規　　格——***Project LIBC Group***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isalnum，isalpha，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper

isatty

用　途——端末デバイスであるかどうかを調べる。

書　式——

```
#include <unistd.h>
int isatty (int fildes);
```

解　説——isatty 関数は，*fildes* で指定したファイルハンドルが指すファイルが端末デバイス (キャラクタデバイス) であるかどうかを調べる。

戻り値——端末デバイスならば 1 を返し，端末デバイスでなければ 0 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- ENOTTY　端末デバイスではない

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——ttyname

isblank

用　　途――ブランク文字かどうかを調べる。

書　　式――#include <ctype.h>

```
int isblank (int c);
```

解　　説――isblank 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**ブランク文字**(スペース) であるかどうかを調べる。

戻 り 値――*c*が記号であれば 0 以外の値を返し，異なれば 0 を返す。

注　　意――isspace 関数との違いに気をつけること。

*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isblank 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。isblank 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　　格――***Project LIBC Group***

関連項目――isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper，isxdigit

iscntrl

用　途——制御文字かどうかを調べる。

書　式——

```
#include <ctype.h>
int iscntrl (int c);
```

解　説——iscntrl 関数は，*c*で指定した文字コードを現在の**ロケール**の LC_CTYPE カテゴリに照らし合わせ，**制御文字 (コントロール文字)** であるかどうかを調べる。

戻り値——*c*が制御文字であれば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，iscntrl 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。iscntrl 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isalnum，isalpha，isascii，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper，isxdigit

isdigit

用　　途――数字かどうかを調べる。

書　　式――#include <ctype.h>
int isdigit (int c);

解　　説――isdigit 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，数字であるかどうかを調べる。

戻 り 値――*c*が数字であれば0以外の値を返し，異なれば0を返す。

注　　意――*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isdigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isdigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はＣロケールしかサポートしていない。

規　　格――*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――isalnum，isascii，iscntrl，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper，isxdigit

isgraph

用　途——表示可能文字かどうかを調べる。

書　式——
```
#include <ctype.h>
int isgraph (int c);
```

解　説——isgraph 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**表示可能文字**（スペースを除く印字可能文字）であるかどうかを調べる。

戻り値——*c*が表示可能文字であれば0以外の値を返し，異なれば0を返す。

注　意——*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isgraph 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isgraph 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isalnum，isalpha，isascii，iscntrl，isdigit，isiso，islower，isodigit，isprint，ispunct，isspace，isupper，isxdigit，

I

isinf

用　　途——無限大かどうかを調べる。

書　　式——

```
#include <math.h>
int isinf (double value);
```

解　　説——isinf 関数は，*value*の数値が無限大(符号は問わない)かどうかを調べる。

戻 り 値——数値が無限大ならば0以外の値を返し，無限大でなければ0を返す。

規　　格——***Project LIBC Group***

関連項目——isnan

isiso

用　途——8 ビット ISO 文字かどうかを調べる。

書　式——

```
#include <ctype.h>
int isiso (int c);
```

解　説——isiso 関数は，*c* で指定した文字コードが 8 ビット ISO 文字 (0x00 ～ 0xFF) であるかどうかを調べる。

戻り値——*c* が 8 ビット ISO 文字であれば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isiso 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。isiso 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

isiso 関数はロケールとは関係がない。

規　格——*Project LIBC Group*

関連項目——isalpha, isascii, iscntrl, isdigit, isgraph, islower, isodigit, isprint, ispunct, isspace, isupper, isxdigit

islower

用　途——英小文字かどうかを調べる。

書　式——#include <ctype.h>

```
#include <ctype.h>
int islower (int c);
```

解　説——islower 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**英小文字**であるかどうかを調べる。

戻り値——*c*が英小文字であれば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，islower 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。islower 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, isiso, isodigit, isprint, ispunct, isspace, isupper, isxdigit

isnan

用　　途——非数かどうかを調べる。

書　　式——#include <math.h>

```
#include <math.h>
int isnan (double value);
```

解　　説——isnan 関数は *value* の数値が非数かどうかを調べる。*LIBC* では Quiet NaN と Signaling NaN の区別はしない。

戻 り 値——数値が非数ならば 0 以外の値を返し，非数でなければ 0 を返す。

規　　格——*Project LIBC Group*，　*XPG3*，　*AES/OS*

関連項目——isinf

isodigit

用　　途——8進数字かどうかを調べる。

書　　式——#include <ctype.h>

```
int isodigit (int c);
```

解　　説——isodigit 関数は，*c* で指定した文字コードが8進数字であるかどうかを調べる。8進数字とは0～7のことを指す。

戻 り 値——*c* が8進数字であれば0以外の値を返し，異なれば0を返す。

注　　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isodigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。isodigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

isodigit 関数はロケールとは関係がない。

規　　格——*Project LIBC Group*，　*MS–C 7.0*

関連項目——isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper

isprint

用　途──印字可能文字かどうかを調べる。

書　式──`#include <ctype.h>`
`int isprint (int c);`

解　説──isprint 関数は，*c*で指定した文字コードを現在の**ロケール**の LC_CTYPE カテゴリに照らし合わせ，**印字可能文字**であるかどうかを調べる。

戻り値──*c*が印字可能文字であれば 0 以外の値を返し，異なれば 0 を返す。

注　意──*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isprint 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isprint 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*は C ロケールしかサポートしていない。

規　格──***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目──isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，ispunct，isspace，isupper，isxdigit

ispunct

用　途――記号文字かどうかを調べる。

書　式――#include <ctype.h>

```
int ispunct (int c);
```

解　説――ispunct関数は，*c*で指定した文字コードを現在のロケールのLC_CTYPEカテゴリに照らし合わせ，**記号文字**であるかどうかを調べる。

戻り値――*c*が記号文字であれば0以外の値を返し，異なれば0を返す。

注　意――*c*はint型の引数だが，その値はunsigned char型で表現できる範囲の値か，あるいはEOFと同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ispunct関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ispunct関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はCロケールしかサポートしていない。

規　格――***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***，***4.3BSD***

関連項目――isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, isiso, islower, isodigit, isspace, isupper, isxdigit

isspace

用　　途——空白文字かどうかを調べる。

書　　式——
```
#include <ctype.h>
int isspace (int c);
```

解　　説——isspace 関数は，*c*で指定した文字コードを現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**空白文字**(スペース/タブ/改行/復帰/垂直タブ/改頁など)であるかどうかを調べる。

戻 り 値——*c*が記号であれば0以外の値を返し，異なれば0を返す。

注　　意——*c*は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isspace 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isspace 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はＣロケールしかサポートしていない。

規　　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isupper，isxdigit

isupper

用　　途——英大文字かどうかを調べる。

書　　式——#include <ctype.h>

```
int isupper (int c);
```

解　　説——isupper関数は，*c*で指定した文字コードを現在のロケールのLC_CTYPEカテゴリに照らし合わせ，英大文字であるかどうかを調べる。

戻 り 値——*c*が英大文字であれば0以外の値を返し，異なれば0を返す。

注　　意——*c*はint型の引数だが，その値はunsigned char型で表現できる範囲の値か，あるいはEOFと同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isupper関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。isupper関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はCロケールしかサポートしていない。

規　　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isxdigit

isxdigit

用　途——16 進数字かどうかを調べる。

書　式——

```
#include <ctype.h>
int isxdigit (int c);
```

解　説——isxdigit 関数は，*c* で指定した文字コードが 16 進数字であるかどうかを調べる。16 進数字とは 0 ～ 9，A ～ F および a ～ f のことを指す。

戻り値——*c* が 16 進数字であれば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値か，あるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，isxdigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。isxdigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

isxdigit 関数はロケールとは関係がない。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——isalnum，isalpha，isascii，iscntrl，isdigit，isgraph，isiso，islower，isodigit，isprint，ispunct，isspace，isupper

kbhit

用　　途——コンソールへの入力の有無を調べる。

書　　式——

```
#include <conio.h>
int kbhit (void);
```

解　　説——kbhit関数はコンソールの状態を調べ，キーボードが押されているか(あるいは入力データがあるか)どうかを調べる。kbhit関数はつねにコンソールの状態を調べるので，標準入力の状態に左右されることもstdioライブラリによるバッファリングの作用も受けない。

kbhit関数は入力を待たないが，すでにキーボードバッファにデータがある場合やungetch関数によって押し戻されたデータがある場合は，それらを入力データとして認識する。

戻り値——キーボードが押されていれば(入力データがあれば)そのデータを返し，なければ0を返す。エラーはない。

注　　意——kbhit関数は入力データがあるかどうかを調べるためのものであり，必ずしもキーボードが押されているかどうかを調べることにはならない。

規　　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——getch，getche

kill

用　途——プロセスに対してシグナルを送信する。

書　式——

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

解　説——kill 関数は，*pid*で指定したプロセスに対して *sig*で指定したシグナルを送信する。ただし，*LIBC*では自プロセスに対するシグナル送信しかサポートしていない。

戻り値——正常にシグナルを発生できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正なシグナル番号 *sig*を指定した
- ESRCH　*pid*で指定したプロセスが存在しない

注　意——自プロセスに対してシグナルを送信する場合には，*pid*に getpid 関数で求めたプロセス ID を指定するか，raise 関数を使用すること。

互換性——**Human68k** はシグナル機構をサポートしていないので，*LIBC*では内部でソフト的に例外を発生させることでシグナル処理を行っている。したがって *LIBC*では，自プロセス以外にシグナルを送信することはできない。

規　格——*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——getpid，raise，sigaction，signal

labs

用　　途──long 型の値の絶対値を取得する。

書　　式──#include <stdlib.h>

```
long labs (long n);
```

解　　説──labs 関数は n の絶対値を返す。

戻 り 値──n の long 型絶対値を返す。

注　　意──通常，labs 関数はマクロとして定義されるが，__NO_STDLIB_INLINE__ が定義された場合には実体をもつ関数となる。

規　　格──*ANSI C*，*SYSV*

関連項目──abs，div，ldiv，wabs

ldexp

用　途——x に 2^{exp} を乗じた値を返す。

書　式——
```
#include <math.h>
double ldexp (double x, int *exp);
#include <sys/xmath.h>
double _f_ldexp (double x, int *exp);
double _fe_ldexp (double x, int *exp);
double _fpu_ldexp (double x, int *exp);
```

解　説——ldexp 関数は x に 2^{exp} を乗じた値を返す。各関数はそれぞれ次のように動作する。

- ldexp　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合はソフトウェアエミュレートする
- _f_ldexp　数値演算コプロセッサ命令を直接ドライブする
- _fe_ldexp　ソフトウェアエミュレートする
- _fpu_ldexp　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合はその値を返す。もし x が非数 (NaN) ならば変数 errno にその原因を示すエラーコードを設定して同じく非数を返し，x が 0 ならば変数 errno にその原因を示すエラーコードを設定して同じく 0 を返し，オーバーフローした場合は変数 errno にその原因を示すエラーコードを設定して HUGE_VAL を返す。

- EDOM　x の値が非数
- ERANGE　x が 0
- ERANGE　計算結果がオーバーフローした

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは ldexp は _fe_ldexp の別名となり，__DIRECT_IOFPU__が定義されているときは ldexp は _fpu_ldexp の別名となる。また，__DIRECT_FPU__が定義されているときは ldexp は _f_ldexp の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——frexp，modf

ldiv

用　　途――long 型整数の除算を実行する。

書　　式――#include <stdlib.h>

```
ldiv_t ldiv (long number, long denom);
```

解　　説――ldiv 関数は，long 型の整数である *number* を *denom* で割った商と剰余を求める。符合は引数の符合が同じであれば正，異なれば負となる。また *denom* で割り切れない場合，計算結果は代数学上の商よりも小さい最大の整数となる。

戻 り 値――除算の結果を ldiv_t 構造体に代入し，その構造体を返す。構造体の定義は次のとおりであり，quot に商を，rem に剰余を格納する。

```
struct {
    long quot;    /* 商 */
    long rem;     /* 剰余 */
};
```

注　　意――変換した結果が long 型で表現できない場合，その結果は不定である。

規　　格――*ANSI C*，*SYSV*

関連項目――div，floor

L

localtime

用　　途――暦時間を地域時間に変換する。

書　　式――#include <time.h>

```
struct tm *localtime (const time_t *timep);
```

解　　説――localtime 関数は，*timep* が指す領域に格納されている**暦時間**（エポックタイム）をタイムゾーンの設定にしたがって**地域時間**に変換し，その結果を tm 構造体に格納してそのアドレスを返す。

localtime 関数は，変換にさきだって tzset 関数を呼び出して地域時間情報を再計算する。tm 構造体は次のとおり。

```
struct tm {
    int   tm_sec;     /* 秒 (0-61) */
    int   tm_min;     /* 分 (0-59) */
    int   tm_hour;    /* 時 (0-23) */
    int   tm_mday;    /* 日 (1-32) */
    int   tm_mon;     /* 月 (0-11) */
    int   tm_year;    /* 年 (年 - 1900) */
    int   tm_wday;    /* 曜日 (0-7 日曜は 0) */
    int   tm_yday;    /* 年内の通算日 (0-365) */
    int   tm_isdst;   /* サマータイム (0-1 実施中なら 1) */
    int   tm_gmtoff;  /* グリニッジから西回り */
    char *tm_zone;    /* タイムゾーン名 */
};
```

戻 り 値――正常に変換できた場合は tm 構造体へのポインタを返し，失敗した場合には NULL を返す。

注　　意――結果は関数内部の静的領域に格納されるため，localtime 関数，gmtime 関数，asctime 関数，ctime 関数のいずれかの関数を呼び出すたびに，内容が上書きされることに注意すること。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――asctime，ctime，gmtime，tzset

locking

用　途——ファイル中の領域ロックの設定/解除を実行する。

書　式——
```
#include <sys/locking.h>
int locking (int fildes, int mode, long n);
```

解　説——locking 関数は，*fildes* で指定したファイルハンドルが指すファイルの現在位置から *n* の領域に対して，ロックを設定あるいは解除する。一度ロックが設定されると，ロックを行ったプロセス以外からは一切読み書きできなくなる。ロック方法は *mode* で指定することができ，以下の値を取る。

- LK_LOCK　指定された領域をロックする。ロックできない場合は 1 秒ごとに 1 回，合計 10 回までリトライする
- LK_RLCK　指定された領域をロックする。ロックできない場合は 1 秒ごとに 1 回，合計 10 回までリトライする
- LK_NBLCK　指定された領域をロックする
- LK_NBRLCK　指定された領域をロックする
- LK_UNLCK　指定された領域に設定されているロックを解除し，他のプロセスからもアクセスできるようにする

戻り値——正常にロック設定/解除できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　不正な *fildes* を指定した
- EINVAL　不正な *mode* を指定した
- EDEADLOCK　10 回リトライしてもロックを設定できない

注　意——1 つのファイル中に複数のロック領域を設定することができるが，それぞれを重複させることはできない。また，領域が隣り合っていても，それぞれ 1 つの領域として扱われることに注意すること。

locking 関数を使用するには，“CONFIG.SYS” に SHARE= を記述してファイルシェアリングができるようにしておく必要がある。

規　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——creat，open

log

用　　途――自然対数 $log(x)$ を求める。

書　　式――
```
#include <math.h>
double log (double x);
#include <sys/xmath.h>
double _f_log (double x);
double _fe_log (double x);
double _fpu_log (double x);
```

解　　説――log関数は x の自然対数(ログ)を求める。各関数はそれぞれ次のように動作する。

- log　　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合はFLOATパッケージを呼び出す
- _f_log　　数値演算コプロセッサ命令を直接ドライブする
- _fe_log　　FLOATパッケージを呼び出す
- _fpu_log　　I/O数値演算コプロセッサを直接ドライブする

戻 り 値――正しく値が求められた場合，log関数は $x > 0$ の範囲で x の自然対数を返す。また，x の値が $x = 0$ だとオーバーフローを起こす。もし x が非数(NaN)ならば変数errnoにその原因を示すエラーコードを設定して同じく非数を返し，それ以外のエラーでは変数errnoにその原因を示すエラーコードを設定して無限大を返す。

- EDOM　　x の値が非数，または x の値が計算範囲外
- ERANGE　　計算結果がオーバーフローした

注　　意――<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときはlogは_fe_logの別名となり，__DIRECT_IOFPU__が定義されているときはlogは_fpu_logの別名となる。また，__DIRECT_FPU__が定義されているときはlogは_f_logの別名となる。

数値演算コプロセッサの場合のみ，ERANGEが設定される。

規　　格――*Project LIBC Group*, *ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目――exp, isnan, log10

log10

用　途——常用対数 $log_{10}(x)$ を求める。

書　式——

```
#include <math.h>
double log10 (double x);
#include <sys/xmath.h>
double _f_log10 (double x);
double _fe_log10 (double x);
double _fpu_log10 (double x);
```

解　説——log10 関数は x の常用対数 (ログ) を求める。各関数はそれぞれ次のように動作する。

- log10　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_log10　数値演算コプロセッサ命令を直接ドライブする
- _fe_log10　FLOAT パッケージを呼び出す
- _fpu_log10　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，log10 関数は $x > 0$ の範囲内で x の自然対数を返す。また，x の値が $x = 0$ だとオーバーフローを起こす。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返し，それ以外のエラーでは変数 errno にその原因を示すエラーコードを設定して無限大を返す。

- EDOM　x の値が非数，または x の値が計算範囲外
- ERANGE　計算結果がオーバーフローした

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは log10 は _fe_log10 の別名となり，__DIRECT_IOFPU__が定義されているときは log10 は _fpu_log10 の別名となる。また，__DIRECT_FPU__が定義されているときは log10 は _f_log10 の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

L

規　格——*Project LIBC Group*, *ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——exp, isnan, log

longjmp

用　　途——大域ジャンプを実行する。

書　　式——#include <setjmp.h>

```
void __volatile longjmp (jmp_buf env, int rval);
```

解　　説——longjmp 関数は，setjmp 関数で jmp_buf 型の配列 *env* に記憶したジャンプポイントに大域ジャンプを行い，その結果，プログラムカウンタは *env* に記憶されたジャンプポイントに移る。一方ジャンプポイントでは，setjmp 関数が見かけ上の戻り値として *rval* を返すように見える。

戻 り 値——なし。longjmp 関数は決して戻ってこず，*env* で指定したジャンプポイントに実行を移す。

注　　意——setjmp 関数での戻り値 *rval* は，記述した場合と longjmp 関数によって制御を移された場合とを区別する必要があるので，0 以外の値でなくてはならない。*rval* に 0 を設定した場合は 1 とみなされる。

env は，あらかじめ setjmp 関数によって環境を格納しておく必要がある。もし格納されていなかった場合，その動作は一切保証されない。

longjmp 関数が実行され，ジャンプポイントの位置に戻った場合，レジスタに割り当てられた auto 型変数を除くアクセス可能なすべてのオブジェクトは，longjmp 関数が実行されたときの値をもつ。

setjmp 関数を実行した関数が return やその他の大域ジャンプなどによって，より上位 (呼び出し元) の関数に戻ってしまった場合，その後で longjmp 関数によってジャンプポイントに戻ろうとしても，その動作は保証されない。

シグナル処理中に発生した別のシグナルの処理の最中に longjmp 関数を実行した場合，すなわち入れ子になったシグナル割り込み処理からの longjmp 関数の動作は保証されない。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——setjmp，sigaction，siglongjmp，sigsetjmp

L

lseek

用　途——ファイルポインタの位置を再設定する。

書　式——
```
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

解　説——lseek関数は，*fildes*で指定したファイルハンドルのファイルポインタの位置を再設定する。新しい位置は*whence*で指定された開始位置に，*offset*バイト分のオフセットを加算したものとなる。*whence*は次のマクロ値を取り，それぞれ開始位置を以下のように定義する。

- SEEK_SET　ファイルの先頭から計算する
- SEEK_CUR　現在のファイルポインタの位置から計算する
- SEEK_END　ファイルの終端から計算する

lseek関数はファイルの先頭を越えることはできないが，終端を越えてファイルサイズを拡張することはできる。この場合，拡張された部分はすべて0で埋められる。

戻り値——正常にシークできた場合はシーク後のファイルポインタの位置を返し，失敗した場合は−1を返して，変数errnoにその原因を示すエラーコードを設定する。このとき，ファイルポインタの位置は変更されない。

- EBADF　不正な*fildes*を指定した
- ENOSPC　ディスクがいっぱいである
- ESPIPE　シークできないファイルを指定した
- EINVAL　不正なオフセットを指定した

互換性——ファイルがテキストモードでオープンされている場合，read関数やwrite関数の戻り値を用いてファイルポインタを計算してはならない。必ずtell関数を用いること。

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——open，tell

_main

用　　途──プログラムのエントリールーチン。

書　　式──#include <sys/xstart.h>

```
void __volatile _main (void);
```

解　　説──_main 関数は，プログラムをリンクしたときにデフォルトで設定されるエントリールーチンである。*LIBC* を使用しているプログラムは，すべてこのルーチンから実行が開始される。

_main 関数は特定の処理はせずに，そのまま_start 関数へ処理を移行させる。つまり，_main 関数は HUPAIR 識別子，ライブラリのバージョン識別子を格納するためにだけ存在する。

_main 関数の先頭から，+2 バイトの位置から 8 バイトに HUPAIR 識別子が，続く 12 バイトにライブラリ識別子が設定されている。

- HUPAIR 識別子　　"#HUPAIR\0"
- ライブラリ識別子　　"#LIBC??????\0"

戻 り 値──なし。_main 関数は決して戻ってこない。

互 換 性──_main 関数の内部処理は *XC* のものとはかなり異なり，既存のライブラリのうちで_main 関数を置き換える必要があるもの，たとえば cshwild，jshwild，SX–Window プログラムなどとは共存させることができない。

もしリンクして無理に共存させた場合，その動作およびそれによって引き起こされる結果は不定であり，一切保証されない。

規　　格──*Project LIBC Group*

関連項目──_start

M

_makepath

用　　途――パスの各要素からパス名を構成する。

書　　式――#include <stdlib.h>

```
void _makepath (char *path, const char *drive, const char *dir,
                const char *fname, const char *ext);
```

解　　説――_makepath 関数は引数で指定されたパス名の各要素から完全なパス名を構成し，その結果を *path* が指す領域に格納する。指定するそれぞれの引数の意味は，次のとおりである。

- *drive*　　ドライブ名。たとえば“A:”など。もしドライブ名の区切り記号であるコロンがない場合は自動的に加えられ，*drive* が NULL の場合は省略される
- *dir*　　ディレクトリ名。たとえば“foo/bar/”など。もし *dir* の最後にパスの区切り記号である“/”や“¥”がない場合は自動的に加えられ，*dir* が NULL の場合は省略される
- *fname*　　ファイル名。たとえば“sample”など拡張子を除いた部分。もし *fname* が NULL の場合は省略される
- *ext*　　ファイルの拡張子。たとえば“.c”など。もし先頭にピリオドがない場合は自動的に加えられ，*ext* が NULL ならば拡張子とピリオドは省略される

戻 り 値――なし。

注　　意――_makepath 関数はパスの構成要素を組み合わせるだけである。それぞれの構成要素やその結果が，ファイル名として有効なものになるかどうかについてはチェックしない。

規　　格――***Project LIBC Group，　MS-C 7.0***

関連項目――_fullentry，_fullpath，_splitpath

変　　更――従来では各引数に NULL ポインタが渡された場合，その部分を省略するようになっていたが，現在の *LIBC* ではこれに加えて空文字でも省略するようにした。

_mode2dos

用　途——ファイルモードを拡張 UNIX アクセスモードから DOS ファイルアトリビュートに変換する。

書　式——#include <sys/xunistd.h>

```
dosmode_t _mode2dos (const mode_t mode);
```

解　説——_mode2dos 関数は *mode* で指定される拡張 UNIX アクセスモードを，以下の表にしたがって DOS ファイルアトリビュートに変換する。

拡張 UNIX アクセスモード	DOS ファイルアトリビュート
S_IREAD	*true*
S_IWRITE	*not* _DOS_IRDONLY
S_IEXEC	_DOS_IEXEC *or* _DOS_IFDIR
S_IFDIR	_DOS_IFDIR
S_IFREG	_DOS_IFREG
S_IFLNK	_DOS_IFLNK
S_IFVOL	_DOS_IFVOL
S_ISYS	_DOS_ISYS
S_IHIDDEN	_DOS_IHIDDEN
S_IEXBIT	_DOS_IEXEC

戻り値——変換した DOS ファイルアトリビュートを返す。

注　意——_mode2dos 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——*Project LIBC Group*

関連項目——_mode2unix

_mode2unix

用　途――ファイルモードをDOSファイルアトリビュートから拡張UNIXアクセスモードに変換する。

書　式――#include <xunistd.h>

```
mode_t _mode2unix (dosmode_t attr);
```

解　説――_mode2unix関数は*attr*で指定されるDOSファイルアトリビュートを，以下の表にしたがって拡張UNIXアクセスモードに変換する。

DOSファイルアトリビュート	拡張UNIXアクセスモード
_DOS_IRDONLY	(*not* S_IWRITE) *or* S_IREAD
_DOS_IHIDDEN	S_IHIDDEN
_DOS_ISYS	S_ISYS
_DOS_IFVOL	S_IFVOL
_DOS_IFDIR	S_IFDIR
_DOS_IFREG	S_IFREG
_DOS_IFLNK	S_IFLNK
_DOS_IEXEC	S_IEXBIT

戻り値――変換した拡張UNIXアクセスモードを返す。

注　意――_mode2unix関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格――*Project LIBC Group*

関連項目――_mode2dos

malloc

用　途——メモリブロックを確保する。

書　式——#include <stdlib.h>

```
void *malloc (size_t size);
```

解　説——malloc 関数は *size* バイトのメモリ領域をヒープ領域から確保し，そのメモリ領域へのポインタを返す。

また，確保されたメモリ領域の先頭アドレスはいかなるデータ型にでもキャスト可能なように，CPU 依存のアラインメントに調整される。

戻り値——正常に確保できた場合はその領域へのポインタを返し，失敗した場合には NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなったか，または制限値に達した

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——brk，calloc，free，rbrk，realloc，sbrk

max

用　　途——2つの最大値を求める。

書　　式——#include <stdlib.h>

```
int max (int a, int b);
#include <sys/param.h>
int MAX (int a, int b);
```

解　　説——max は *a* と *b* を比較し，どちらか大きいほうの値を返す。max はマクロとして定義されるので，値として比較できる変数型であればなんでも比較することができる。そのため，上記の int 型以外にもいくらでも組み合わせが考えられる。ただし，*a* と *b* の変数型が異なる場合は，キャストによる影響を考えなければならない。その場合，*a* と *b* のどちらの変数型にキャストしてから比較されるかは，*ANSI C* の規定およびコンパイラによる。

戻 り 値——*a* と *b* とを比較して，大きいほうの値を返す。もし *a* と *b* の型が異なる場合，戻り値は大きいほうの値の変数型になる。

注　　意——max はつねにマクロとして定義される。また，MAX はすべての面で max と同じである。

規　　格——***Project LIBC Group***

関連項目——MIN，min

mblen

用　　途──1マルチバイト文字の構成バイト数を調べる。

書　　式──#include <stdlib.h>

```
int mblen (const char *s, size_t n);
```

解　　説──mblen 関数は s で指定した領域の先頭の n バイトに含まれる1マルチバイト文字が，何バイトで構成されているかについて調べる。

mblen 関数の動作は，ロケールの LC_CTYPE カテゴリによって変化する。もし現在のロケールの文字コードが状態依存体系であれば，s に NULL を指定することで，シフト状態を初期化することができる。

戻 り 値──s が正しいマルチバイト文字を指している場合は，その構成バイト数を返し (ただし null 文字は0とする)。不正なマルチバイト文字を指していたり，n バイト以上のマルチバイト文字を指している場合は-1を返して，変数 errno にその原因を示すエラーコードを設定する。

また s に NULL を指定した場合は，現在のロケールの文字コードが状態依存体系であれば0以外の値を，状態依存体系でなければ0を返す。

- EINVAL　s が不正なマルチバイト文字を指している

注　　意──*LIBC* はＣロケールしかサポートしていない。

規　　格──*Project LIBC Group*，*ANSI C*，*AES/OS*，*SYSV*

mbstowcs

用　途——マルチバイト文字列を幅広文字列に変換する。

書　式——#include <stdlib.h>

```
#include <stdlib.h>
size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
```

解　説——mbstowcs 関数は *s* で指定されたマルチバイト文字列を幅広文字列に変換し，結果を *pwcs* が指す領域に格納する。変換は null 文字に到達するか，*n* 幅広文字を格納するまで行われる。

変換はロケールの LC_CTYPE カテゴリに影響される。もし現在のロケールが状態依存体系の文字コードを用いていた場合，変換はシフト状態を初期化してから行われる。

戻り値——変換して格納された幅広文字数(終端の null 文字は含まない)を返す。ただし，途中で不正なマルチバイト文字に出会った場合変換はそこで中止され，−1 を返して変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　変換中に不正なマルチバイト文字に出会った

注　意——*LIBC* は C ロケールしかサポートしていない。

規　格——***Project LIBC Group***，***ANSI C***，***AES/OS***，***SYSV***

関連項目——mblen，mbtowc，wcstombs，wctomb

mbtowc

用　途――マルチバイト文字を幅広文字に変換する。

書　式――#include <stdlib.h>

```
int mbtowc (wchar_t *pwc, const char *s, size_t n);
```

解　説――mbtowc 関数は *s* で指定された領域の先頭 *n* バイトに含まれる 1 マルチバイト文字を幅広文字に変換し，結果を *pwc* が指す領域に格納する。

変換はロケールの LC_CTYPE カテゴリに影響される。もし現在のロケールが状態依存体系の文字コードを用いていた場合，*s* に NULL を指定することで，シフト状態を初期化することができる。

戻り値――*s* が正しいマルチバイト文字を指している場合は，その構成バイト数を返す (ただし null 文字は 0 とする)。不正なマルチバイト文字を指していたり，*n* バイト以上のマルチバイト文字を指している場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

また *s* に NULL を指定した場合は，現在のロケールの文字コードが状態依存体系であれば 0 以外の値を，状態依存体系でなければ 0 を返す。

- EINVAL　変換中に不正なマルチバイト文字に出会った

注　意――*LIBC* は C ロケールしかサポートしていない。

規　格――*Project LIBC Group*，*ANSI C*，*AES/OS*，*SYSV*

関連項目――mblen，mbstowcs，wcstombs，wctomb

M

memccpy

用　　途――指定文字まで領域をコピーする。

書　　式――#include <string.h>

```
void *memccpy (void *region1, const void *region2,
               int c, size_t n);
```

解　　説――memccpy 関数は *region2* の指す領域から *n* バイト，あるいは最初に見つかった *c* までを *region1* の指す領域にコピーする。memccpy 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。*c* は，内部で int 型から unsigned char 型に変換される。

戻 り 値――*region1* へのポインタを返す。

注　　意――領域が重なっていた場合の動作は未定義である。また，*region1* は *n* バイトのデータを格納するのに十分な領域を指していなければならない。1文字は1バイトとなる。

規　　格――*XPG3*，　*AES/OS*，　*SYSV*

関連項目――memcpy，memmove，strcpy，strncpy

memchr

用　途――領域中から指定文字を検索する。

書　式――#include <string.h>

```
void *memchr (const void *region, int character, size_t n);
```

解　説――memchr 関数は *region* の指す領域から *n* バイトまで，*character* で指定された文字を検索する。memchr 関数は strchr 関数とは異なり，null 文字を検出しても処理を中断しない。*character* は，内部で int 型から unsigned char 型に変換される。

戻り値――領域中で最初に現れた *character* へのポインタを返し，見つからない場合は NULL を返す。

注　意――1文字は1バイトとなる。

規　格――*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――strchr，strcspn，strpbrk，strrchr，strspn，strstr，strtok

M

memcmp

用　途――2つの領域の内容を比較する。

書　式――#include <string.h>

```
int memcmp (const void *region1, const void *region2, size_t n);
```

解　説――memcmp関数は，*region1*の指す領域と*region2*の指す領域の内容を*n*バイト比較する。memcmp関数はstrcmp関数とは異なり，null文字を検出しても処理を中断しない。

戻り値――比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での*region1*側の文字が*region2*側の文字よりも大きいならば正の値を，小さいならば負の値を返す。

注　意――1文字は1バイトとなる。

規　格――*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――strcmp，strcoll，strncmp，strxfrm

memcpy

用　途——領域をコピーする。

書　式——#include <string.h>

```
void *memcpy (void *region1, const void *region2, size_t n);
```

解　説——memcpy 関数は *region2* の指す領域から *n* バイトを，*region1* の指す領域にコピーする。memcpy 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。

戻り値——*region1* へのポインタを返す。

注　意——領域が重なっていた場合の動作は未定義である。また，*region1* は *n* バイトのデータを格納するのに十分な領域を指していなければならない。1 文字は 1 バイトとなる。

規　格——*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目——memmove，strcpy，strncpy

memmove

用　　途――領域をコピーする。

書　　式――#include <string.h>
void *memmove (void *region1, void *region2, size_t n);

解　　説――memmove 関数は *region2* の指す領域から *n* バイトを，*region1* の指す領域にコピーする。memmove 関数は strcpy 関数とは異なり，null 文字を検出しても処理を中断しない。また memcpy 関数とは異なり，*region1* と *region2* の指す領域が重なっていても正しくコピーできる。

戻り値――*region1* へのポインタを返す。

注　　意――*region1* は，*n* バイトのデータを格納するのに十分な領域を指していなければならない。1 文字は 1 バイトとなる。

規　　格――*ANSI C*，*AES/OS*，*SYSV*

関連項目――memcpy，strcpy，strncpy

memset

用　途――領域を指定文字で埋める。

書　式――#include <string.h>

```
void *memset (void *region, int character, size_t n);
```

解　説――memset 関数は *region* の指す領域から *n* バイトを，*character* で指定された文字で埋める。memset 関数は strset 関数とは異なり，null 文字を検出しても処理を中断しない。*character* は，内部で int 型から unsigned char 型に変換される。

戻り値――*region* へのポインタを返す。

注　意――1 文字は 1 バイトとなる。

規　格――*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――memchr，memcmp，memcpy，memmove

min

用　　途——2つの最小値を求める。

書　　式——
```
#include <stdlib.h>
int min (int a, int b);
#include <sys/param.h>
int MIN (int a, int b);
```

解　　説——min は a と b を比較し，どちらか小さいほうの値を返す。min はマクロとして定義されるので，値として比較できる変数型であればなんでも比較することができる。そのため，上記の int 型以外にもいくらでも組み合わせが考えられる。ただし a と b の変数型が異なる場合は，キャストによる影響を考えなければならない。その場合，a と b のどちらの変数型にキャストしてから比較されるかは，***ANSI C*** の規定およびコンパイラによる。

戻 り 値——a と b とを比較して，小さいほうの値を返す。もし a と b の型が異なる場合，戻り値は小さいほうの値の変数型になる。

注　　意——min はつねにマクロとして定義される。また，MIN はすべての面で min と同じである。

規　　格——***Project LIBC Group***

関連項目——MAX，max

mkdir

用　　途——ディレクトリを作成する。

書　　式——#include <sys/stat.h>

```
int mkdir (const char *path, mode_t mode);
```

解　　説——mkdir 関数は *path* で指定したディレクトリを作成する。作成されたディレクトリのファイルモードは，*mode* をプロセスのカレントファイルマスクでマスクした値となる。ファイルモードについての詳細は，chmod 関数あるいは stat 関数を参照のこと。

戻 り 値——正常に作成できた場合は 0 を返し，失敗した場合には–1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EEXIST　ディレクトリがすでに存在する
- ENOENT　不正なパス名を指定した
- ENOSPC　ディスクがいっぱいである
- EROFS　リードオンリーファイルシステムである
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——chmod，stat，umask

M

mktemp

用　　途――テンポラリファイル名を作成する。

書　　式――
```
#include <stdlib.h>
char *mktemp (char *template);
```

解　　説――mktemp 関数はテンプレートをもとにユニークなファイル名を作る。*template*には固定のファイル名部分に，最大6つまでの“X”(置き換え対象文字)をつなげたテンプレート文字列を設定する。この置き換え対象文字列の部分は，mktemp 関数を呼び出した時点での現在時刻を数値化したもので置き換えられる。たとえば，mktemp 関数を呼び出した時点の時刻が7482であった場合，結果は次のようになる。

“/tmp/fileXXXXXX” → “/tmp/file007482”

ただし置き換え対象となりうるのは，固定ファイル名の後ろの連続した“X”であり，以下のような指定をしても，すべての“X”が変換されるわけではない。

“/tmp/fileXYZXYZXXX” → “/tmp/fileXYZXYZ482”

また，mktemp 関数は作成したファイル名が存在するかどうかをチェックし，もし重なった場合は最も先頭にある“X”の部分をa，b，c ...の順番で変化させる。ただし，この部分がzを越えてもユニークにならない場合はエラーとなる。

戻 り 値――正常にファイル名を作成できた場合は*template*を返し，失敗した場合は−1を返す。

注　　意――同じテンプレートを用いて連続してmktemp 関数を呼び出した場合は，ユニークな名前にならない場合がある。

規　　格――***4.3BSD***

関連項目――tmpfile，tmpnam

mktime

用　途——地域時間を暦時間に変換する。

書　式——#include <time.h>

```
time_t mktime (struct tm *timeptr);
```

解　説——mktime 関数は，*timeptr* が指す領域に格納されている tm 構造体によって表される地域時間を暦時間 (エポックタイム) に変換し，その値を返す。

変換にあたっては tm_wday と tm_yday は無視される。また，地域時間を協定世界時間に変換する際に，tm_isdst の値によって変換が調整される。もし tm_isdst が正ならば，現在の日付に関係なく夏時間 (サマータイム) が適用されているものとして，また 0 ならば適用されていないとして計算される。tm_isdst が負ならば，夏時間が適用されるかどうかはタイムゾーンの設定による。

mktime 関数は変換前に，tzset 関数を呼び出して地域時間情報を再計算する。変換後，決定された日時から tm_yday，tm_wday が再計算される。

戻り値——正常に変換できた場合は暦時間を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な地域時間を指定した

注　意——変換が失敗した場合，*timeptr* が指す領域の内容は破壊されることがある。また tm_sec は閏秒を調整するために，0 から 61 まで設定することができる。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——gmtime，localtime，time，tzset

M

modf

用　途——整数部と小数部にわける。

書　式——#include <math.h>

```
double modf (double x, double *iptr);
#include <sys/xmath.h>
double _f_modf (double x, double *iptr);
double _fe_modf (double x, double *iptr);
double _fpu_modf (double x, double *iptr);
```

解　説——modf 関数は x を整数部と小数部に分解し，整数部は *iptr* が指す領域に格納し，小数部を返す。符号は整数部，小数部ともに x と同じになる。各関数はそれぞれ次のように動作する。

- modf　　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_modf　　数値演算コプロセッサ命令を直接ドライブする
- _fe_modf　　FLOAT パッケージを呼び出す
- _fpu_modf　　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合は小数部の値を返す。もし x が非数 (NaN) だった場合は，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　x の値が非数

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは modf は _fe_modf の別名となり，__DIRECT_IOFPU__が定義されているときは modf は _fpu_modf の別名となる。また，__DIRECT_FPU__が定義されているときは modf は _f_modf の別名となる。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——frexp，ldexp

nice

用　途――カレントプロセスの優先度を変更する。

書　式――
```
#include <unistd.h>
int nice (int incre);
```

解　説――nice 関数はカレントプロセスの優先度に *incre* を加算する。プロセスの優先度は $0 \leq x \leq 2 \times$NZERO-1 の値で，プロセスの起動時に自動的に NZERO が設定される。

戻り値――正常に変更できた場合は，新しい優先度 $x-$NZERO を返し，失敗した場合には-1 を返す。

互換性――**Human68k** はマルチタスクではなく，**プロセス** ID **優先度**という概念がないので，nice 関数がこれらの値に影響されることはない。また，***LIBC***内ではデフォルトの**仮想ユーザ** ID および**仮想グループ** ID を root と仮定しているため，nice 関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　格――***Project LIBC Group***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

offsetof

用　　途――構造体メンバのオフセットを求める。

書　　式――#include <stddef.h>

```
int offsetof (structure, member);
```

解　　説――offsetof は，*structure* で指定した構造体の *member* メンバに対する構造体の内部オフセットを求める。

戻 り 値――オフセット値 (正の値) を返す。

規　　格――*ANSI C*

onexit

用　途——プロセス終了時に必ず呼び出される関数を登録する。

書　式——

```
#include <stdlib.h>
onexit_t onexit (onexit_t func);
```

解　説——onexit 関数は *func* で指定した関数を，プロセスの終了時に呼び出される関数のリストに登録する。これら登録された関数は，プロセスが終了するとき (正常終了，異常終了を問わない) に，登録された順番とは逆の順番 (つまり最後に登録した順番) で呼び出される。

プロセスの終了とは，exit 関数によるプロセスを終了，main 関数からの return，abort 関数での終了，エラーによる強制終了などすべての場合を含む。なお登録できる関数の数は，最大 ONEXIT_MAX 個までである。ONEXIT_MAX は<limits.h>に定義されている。

戻り値——正常に登録できた場合は *func* を返し，失敗した場合には NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　すでに ONEXIT_MAX 個の関数が登録されているので，これ以上登録できない

注　意——onexit 関数は割り込みベクタを元に戻す，チップを制御するなど，後処理をせずに終了したのではシステムをパニックに陥れるような重要な処理を登録するのに使用するべきである。その他の一般的な処理については，atexit 関数を使用するほうが望ましい。

互換性——onexit 関数は，*MS-C 7.0* などでは atexit 関数と同じ意味で使用されており，*LIBC* のように重要な処理にかぎられたものではなく，*LIBC* の onexit 関数とは名前は同じだが別物と考えたほうがよい。そのため，onexit 関数を用いたプログラムとはやや互換性を失うので注意すること。

規　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——abort，atexit，exit，sysconf

open

用　途——ファイルをオープンする。

書　式——#include <fcntl.h>

```
int open (const char *path, int oflag, ...);
```

解　説——open 関数は *path* で指定したファイルをオープンし，ファイルハンドルを返す。低水準入出力ではこのファイルハンドルを通して，すべての操作が行われる。

オープン動作は *oflag* で指定したオープンモードにしたがって行われ，ファイルの切り詰めや新規作成などが行われる。オープンモードは次に示した各値のうち，必要なものの論理和をとって指定する。

- O_RDONLY　ファイルを読み込み専用モードでオープンする。ファイルへの書き込みは行うことができない
- O_WRONLY　ファイルを書き込み専用モードでオープンする。ファイルからの読み込みは行うことができない
- O_RDWR　ファイルを読み書き可能モードでオープンする。ファイルへの書き込み，読み込みの両方を行うことができる
- O_TEXT　テキストモードでオープンする。ファイルへの書き込み/読み込みに対してつねに CRLF と LF の間の変換が行われる。O_TEXT，O_BINARY のどちらも指定しない場合は，O_TEXT がデフォルトとなる
- O_BINARY　バイナリモードでオープンする。ファイルへの書き込み/読み込みはすべて **Human68k** と直接行われ，一切データの変換は行われない
- O_APPEND　追加書き込みモードでオープンする。O_WRONLY や O_RDWR といっしょに指定すると，ファイルへの書き込みはつねにファイルの終端に対して追加される
- O_CREAT　open 関数は *path* で指定したファイルが存在しない場合はエラーとなるが，このフラグを指定しておくとファイルが見つからない場合に新規にファイルを作成するようになる。新規に作成されるファイルのファイルアクセスモードは *oflag* の次に指定される mode_t 型の引数 *mode* にカレントファイルマスクを適用した値で設定されることになる
- O_EXCL　O_CREAT といっしょに指定すると，*path* で指定したファイルがすでに存在する場合にはエラーとなる。その結果，つねに新規作成だけが行われるようになり，すでにあるファイルがある場合には破壊しないようにすることができる

- O_TRUNC　　O_WRONLYやO_RDWRといっしょに指定すると，*path*で指定したファイルがすでに存在する場合にはそのファイルの内容を破棄し，サイズを0バイトに切り詰めるようになる

戻 り 値——正常にオープンできた場合はファイルハンドルを返し，失敗した場合は−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EEXIST　*path*で指定したファイルと同名のファイルがすでに存在する
- EISDIR　ディレクトリである
- EMFILE　これ以上ファイルをオープンできない
- ENOENT　*path*で指定したファイルが存在しない
- ENOSPC　ディスクがいっぱいである
- EINVAL　不正な*oflag*を指定した
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

互 換 性——デフォルト状態ではテキストモードでオープンされるので，UNIXのようにテキストモードとバイナリモードの区別がない処理系からプログラムを移植する場合には，明示的にO_BINARYを指定するか，setmode関数でデフォルトのモードを変更しておくこと。

規　　格——***Project LIBC Group***，***MS-C 7.0***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——chmod，close，creat，dup，fcntl，lseek，read，umask，write

opendir

用　途——ディレクトリストリームをオープンする。

書　式——#include <dirent.h>

```
DIR *opendir (const char *name);
```

解　説——opendir 関数は，*name* の指すディレクトリをオープンしてディレクトリストリームを作成し，そのディレクトリストリームへのポインタを返す。

戻り値——正常にディレクトリがオープンできた場合はディレクトリストリームの構造体のポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ディレクトリが見つからない
- ENOTDIR　ディレクトリではない
- ENOMEM　メモリが足りなくなった

注　意——オープンされた後に，そのディレクトリに新規作成されたファイルは対象にはならない。操作対象にするためには再度オープンする必要がある。

規　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——closedir，readdir，rewinddir，seekdir，telldir

PRAMREG

用　途——パラメータレジスタ渡しのためのレジスタを指定する。

書　式——#include <interrupt.h>

```
PRAMREG (int var, reg);
```

解　説——PRAMREG はパラメータを**レジスタ渡し**で行うために，*var* に *reg* で指定したレジスタを割り当て，正しくコンパイルされるようにする。つまりここで指定したレジスタは，関数内部でのレジスタの退避/復帰の候補からはずされるため，外部関数とこのレジスタをとおして値をやりとりすることができるようになる。

```
PRAMREG (_a6_argument, "a6");
```

注　意——PRAMREG は変数の宣言を行うマクロなので，文法的に変数宣言を行える場所にしか記述することはできない。

互 換 性——この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された **GCC** でしか使用することができない。移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——RETREG，SET_FRAME

P

_print

用　　途——コンソールに簡易フォーマットで文字列を出力する。

書　　式——#include <sys/xglob.h>

```
void _print (const char *format, ...);
```

解　　説——_print 関数は *format* で指定されたフォーマットにしたがい，与えられた文字列/データを直接，コンソールに出力する。このため，たとえ標準出力がリダイレクトされても，必ず画面に出力することができる。

format に指定するフォーマット文字列は printf 関数に似ているが，非常に簡略化されたものである。_print 関数は *format* で指定されたフォーマット文字列を画面に出力するが，途中で "%d", "%s", "%x" というフォーマット指定文字に出会うと，それに対応するデータを引数から取り出して表示する。

フォーマット指定文字として使用できるのは次のとおり。

- %d　符号なし 10 進整数を表示する
- %x　符号なし 16 進整数を表示する
- %s　文字列を左詰めで表示する

戻 り 値——なし。

注　　意——_print 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格——***Project LIBC Group***

関連項目——fprintf，printf

pathconf

用　途——パス名に関する情報を取り出す。

書　式——#include <unistd.h>

```
long pathconf (const char *path, int name);
```

解　説——pathconf 関数は *path* で指定したパスに関連する情報を取得し，ユーザアプリケーションに対して現在のシステム設定値などを得る手段を与える。どの設定値を調べるかは *name* で指定する。*name* に指定することができるマクロ値と求める設定値の一覧は次のとおり。

- _PC_LINK_MAX　ハードリンクの最大値 (参考値)
- _PC_MAX_CANNON　端末の Canonical 入力の最大バイト数 (参考値)。*path* は端末デバイスを指していなければならない
- _PC_MAX_INPUT　端末の入力キューの最大バイト数 (参考値)。*path* は端末デバイスを指していなければならない
- _PC_NAME_MAX　ファイル名に許される最大の長さ (ただし null 文字は含まない)
- _PC_PATH_MAX　パス名に許される最大の長さ (ただし null 文字は含まない)
- _PC_PIPE_BUF　パイプバッファの最大バイト数 (*LIBC* では疑似パイプしかサポートされていないので参考値にすぎない)
- _PC_CHOWN_RESTRICTED　chown 関数の使用が制限されているかどうか (*LIBC* では chown 関数は必ず成功するので無制限)
- _PC_NO_TRUNC　NAME_MAX よりも長いファイル名を適正な長さまで切り詰めるかどうか (*LIBC* では一切考慮していないので，切り詰めはない)
- _PC_VDISABLE　端末の特殊文字を無効にすることができる文字 (参考値)。*path* は端末デバイスを指していなければならない

戻り値——正常に制限値を取得できた場合はその値 (無制限ならば−1) を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な *name* を指定したか，または *name* に関する制限値が求められない
- ENOENT　存在しないファイル名を指定した

P

- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

互換性――*LIBC*では上記(詳細は<limits.h>を参照のこと)の制限値は可変ではなく，すべて固定である。

規　格――*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目――fpathconf，sysconf

pause

用　　途——シグナルを受信するまでプロセスの実行を中断する。

書　　式——#include <unistd.h>

```
int pause (void);
```

解　　説——pause 関数は，カレントプロセスに対してシグナルが配信されかつそのシグナルが無視されずにユーザ定義のシグナルハンドラが起動されるか，あるいは強制的に停止させられるまでカレントプロセスの実行を中断する。

戻 り 値——シグナルが配信されるまで永遠に実行を中断するので，正常に処理を完了することはない。シグナルによって処理を再開した場合には–1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINTR　　シグナルによって割り込まれた

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——sigaction，signal，sigsuspend

P

perror

用　途——標準出力にエラーメッセージを出力する。

書　式——#include <stdio.h>

```
void perror (const char *s);
```

解　説——perror 関数は，その時点での変数 errno の値に対応するエラーメッセージを標準出力に出力する。その際，もし *s* が NULL でなくかつ *s* が指す文字列が空文字列でなければ，エラーメッセージの前に，*s* とコロンとスペースを付加した文字列を出力する。

また変数 errno からエラーメッセージへの変換は，strerror 関数を用いて行われる。たとえば変数 errno が ENOENT のときは，*s* に sample を指定して perror 関数を呼び出すと，結果として以下の文字列が出力される。

```
sample: File or directory not found.
```

戻り値——なし。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——strerror

pow

用　途——累乗 x^y を求める。

書　式——
```
#include <math.h>
double pow (double x, double y);
```

解　説——pow 関数は累乗 x^y を求める。

戻り値——正しく値が求められた場合，pow 関数は $x > 0$ の範囲でその値を返す。もし x が非数 (NaN) ならば，変数 errno にその原因を示すエラーコードを設定して同じく非数を返す。

- EDOM　　x の値が非数，または x の値が計算範囲外

注　意——つねに FLOAT パッケージを呼び出す。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——exp，isnan，log，log10

printf

用　　途――標準出力ファイルストリームにフォーマット出力を行う。

書　　式――#include <stdio.h>

```
int printf (const char *format, ...);
```

解　　説――printf 関数は，*format* で指定したフォーマット文字列にしたがって引数を文字列に変換し，結果を**標準出力**に割り当てられた**ファイルストリーム**に出力する。

フォーマット文字列の指定方法や詳細な説明については，fprintf 関数を参照のこと。

戻 り 値――正常に出力できた場合は出力した文字数を返し，失敗した場合は EOF を返してストリームの**エラー指示子**を設定する。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　　格――***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目――eprintf，fprintf，sprintf

psignal

用　途――標準出力にシグナルメッセージを出力する。

書　式――#include <stdio.h>

```
#include <stdio.h>
#include <signal.h>
void psignal (int sig, const char *s);
```

解　説――psignal 関数は，*sig* で指定したシグナルに対応するメッセージを**標準出力**に出力する。その際，もし *s* が NULL でなくかつ *s* が指す文字列が**空文字列**でなければ，エラーメッセージの前に，*s* とコロンとスペースを付加した文字列を出力する。

また *sig* からシグナルメッセージへの変換は，strsignal 関数を用いて行われる。たとえば *sig* が SIGINT のときは，*s* に sample を指定して psignal 関数を呼び出すと，結果として以下の文字列が出力される。

```
sample: SIGINT Interrupted
```

戻り値――なし。

規　格――*Project LIBC Group*，*4.3BSD*

関連項目――strsignal

P

putc

用　　途——ファイルストリームにバイトデータを出力する。

書　　式——

```
#include <stdio.h>
int putc (int c, FILE *stream);
```

解　　説——putc 関数は *stream* で指定されたファイルストリームに対して，文字 *c* を unsigned char 型に変換して出力する。これに応じて，ストリームのファイルポインタは1バイト進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形で行われる。

putc 関数は更新モードで正しく動作しない点を除けば，fputc 関数と同じである。

戻り値——正常に出力できた場合は *c* を返し，何らかのエラーによって失敗した場合はストリームのエラー指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　　意——通常，putc 関数はマクロとして定義されるが，__NO_STDIO_INLINE__ が定義された場合は実体をもつ関数となる。putc 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——fputc

putch

用　途——コンソールへ直接1文字を出力する。

書　式——

```
#include <conio.h>
int putch (int c);
```

解　説——putch関数は，*c*で指定した文字をコンソールに直接表示する。putch関数は，機能的にはputchar関数と似ているが，stdioライブラリによるバッファリングの作用は一切受けない。

また出力はつねにコンソールに対して行われるので，標準出力/標準エラー出力などの状態に左右されることもテキスト/バイナリモードの区別もないので，CRLFをLFに変換する作業も行われない。

戻り値——*c*を返す。エラーはない。

規　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——cprintf，cputs，fputc，putc，putchar

P

putchar

用　途――標準出力ファイルストリームにバイトデータを出力する。

書　式――
```
#include <stdio.h>
int putchar (int c);
```

解　説――putchar 関数は標準出力に割り当てられたファイルストリームに対して，文字 c を unsigned char 型に変換して出力する。これに応じて，ストリームのファイルポインタは1バイト進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

putchar 関数は更新モードで正しく動作しない点を除けば，fputc 関数に stdout を指定した場合と同じである。

戻り値――正常に出力できた場合は c を返し，何らかのエラーによって失敗した場合は，標準出力ストリームのエラー指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　意――通常，putchar 関数はマクロとして定義されるが，__NO_STDIO_INLINE__ が定義された場合は実体をもつ関数となる。putchar 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

規　格――*ANSI C*， *POSIX.1*， *XPG3*， *AES/OS*， *4.3BSD*， *SYSV*

関連項目――fputc，putc

putenv

用　途——環境変数を登録/変更/削除する。

書　式——
```
#include <stdlib.h>
int putenv (const char *string);
```

解　説——putenv 関数は *string* で指定した環境変数を，プロセスの**環境変数テーブル**に対して登録，変更あるいは削除する。ここで，*string* は次の形式をとる。

“name=value”

- *name* で指定した環境変数が環境変数テーブルにない場合は，新たに *name* という名前で環境変数を登録し，その値として *value* を設定する
- *name* で指定した環境変数が環境変数テーブルにすでに存在している場合は，その環境変数の値を *value* で置き換える
- *value* が空文字列だった場合，すなわち *string* が “=” で終わっている場合は環境変数テーブルから *name* という名前の環境変数を削除する

戻り値——成功した場合は 0 を返し，失敗した場合には 0 以外の値を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　意——操作される環境変数テーブルはプロセス内部にコピーされた環境変数テーブルであり，親プロセスの環境変数テーブルや **Human68k** の環境変数テーブルは変更されない。

putenv 関数を実行すると変数 environ が変化するため，以前のポインタ値に対するアクセスは保証されない。また，*LIBC* のなかには環境変数を参照する関数もあるが，参照がどの時点で行われるかは関数によるので，必ずしもクリアすることによって影響を受けるとは限らない。

規　格——***Project LIBC Group***，***XPG3***，***SYSV***

関連項目——clearenv，environ，getenv

P

puts

用　　途――標準出力ファイルストリームに文字列を出力する。

書　　式――#include <stdio.h>

```
int puts (const char *s);
```

解　　説――puts 関数は標準出力に割り当てられたファイルストリームに対して，*s*が指す文字列と改行文字を出力する。文字列は null 文字までとし，最後の null 文字は出力しない。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

puts 関数は fputs 関数とは異なり，文字列の最後に改行文字を付加して出力する。

戻 り 値――正常に出力できた場合は負ではない値を返し，失敗した場合は EOF を返してストリームのエラー指示子を設定する。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――ferror，fopen，fputs，putc

putw

用　途——ファイルストリームにワードデータを出力する。

書　式——#include <stdio.h>

```
int putw (int w, FILE *stream);
```

解　説——putw 関数は，*stream* で指定されたファイルストリームに対してワードデータ *w* を出力する。ワードデータのサイズは int 型を表現する 4 バイトロングワードであり，このサイズはマシンに依存する。また，putw 関数はファイルストリームに対してアラインメントの計算は行わない。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

戻り値——正常にワードデータを出力できた場合は 0 を返し，失敗した場合は 0 以外の値を返して，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

互換性——*XC* では出力データは 2 バイトショートワードだが，*LIBC* では本来の定義どおり 4 バイトロングワードとなっている。したがって，putw 関数を使用したプログラムとは互換性を失う。

規　格——*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——fopen，fwrite，getw

P

qsort

用　　途――配列をクイックソートによって整列させる。

書　　式――

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compare) (const void *, const void *));
```

解　　説――qsort 関数は *base* で指定された配列を，クイックソートアルゴリズムによってソートする。配列は *base* から始まる *nmemb* 個の項目 (1 項目は *size* バイト) であり，比較関数 *compare* の仕様にしたがって正順にソートされる。

qsort 関数はソートを実行するに当たり，必要に応じて任意の 2 項目を関数 *compare* を用いてその大小を比較する。*compare* で指定した関数は，与えられた 2 項目を任意のアルゴリズムで比較し，1 つ目の項目のほうが大きければ正の値，2 つ目の項目のほうが大きければ負の値を，等しければ 0 を返す必要がある。

戻 り 値――なし。

注　　意――ソートの結果，等しい値の項目が並ぶ順番は未定義であり，状況によって変化する。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――bsearch

サンプル――List 1-4 ● 比較関数の例

```
/*
** 任意の 2 つの文字列を比較するための比較関数の例
** compare_str (const void *data1, const void *data2);
** 1 番目の項目が大きければ正，逆なら負，等しければ 0 を返す。
*/

int compare_str (const void *data1, const void *data2)
{
    const char **s1 = (const char **) data1;
    const char **s2 = (const char **) data2;
    return strcmp (*s1, *s2);
}
```

RETREG

用　途——パラメータレジスタ渡しのためのレジスタを指定する。

書　式——#include <interrupt.h>

```
RETREG (int var, reg);
```

解　説——RETREG は戻り値を**レジスタ渡し**で行うために，*var* に *reg* で指定したレジスタを割り当て，正しくコンパイルされるようにする。つまりここで指定したレジスタは，関数内部でのレジスタの退避/復帰の候補からはずされるため，外部関数とこのレジスタを通して値をやりとりすることができるようになる。

```
RETREG (_a6_retvalue, "a6");
```

注　意——RETREG は変数の宣言を行うマクロなので，文法的に変数宣言を行える場所にしか記述することはできない。

互換性——この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された **GCC** でしか使用することができない。移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——PRAMREG，SET_FRAME

R

raise

用　　途――自プロセスに対してシグナルを発行する。

書　　式――#include <signal.h>

```
int raise (int sig);
```

解　　説――raise関数は現在実行中のプログラム(自プロセス)に対して，*sig*で指定したシグナルを配信する。

戻 り 値――正常にシグナルを発生できた場合は0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　　不正なシグナル番号*sig*を指定した

規　　格――*ANSI C*，*AES/OS*，*SYSV*

関連項目――kill, sigaction, signal

rand

用　途——整数の乱数を生成する。

書　式——#include <stdlib.h>

```
int rand (void);
```

解　説——rand 関数は $0 \leq x \leq$ RAND_MAX の範囲で整数の乱数を生成し，その値を返す。

戻り値——$0 \leq x \leq$ RAND_MAX 範囲内の乱数を int 型で返す。

注　意——乱数シードに何も設定しない場合は，デフォルトとしてシードに 1 が用いられる。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——drand，random，srand，srandom

R

random

用　　途――整数の乱数を生成する。

書　　式――#include <stdlib.h>

```
unsigned int random (void);
```

解　　説――random 関数は $0 \leq x \leq$ ULONG_MAX の範囲で整数の乱数を生成し，その値を返す。

戻 り 値――$0 \leq x \leq$ ULONG_MAX 範囲内の乱数を unsigned int 型で返す。

注　　意――乱数シードに何も設定しない場合は，デフォルトとしてシードに1が用いられる。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――drand，rand，srandom

rbrk

用　途——ブレーク値をリセットする。

書　式——
```
#include <stdlib.h>
int rbrk (void);
```

解　説——rbrk 関数は現在のブレーク値を，ヒープ領域が 0 バイトになるような値にリセットする。

ブレーク値についての詳細な説明は brk 関数を参照のこと。

戻 り 値——正常にブレーク値を変更できた場合は 0 を返し，失敗した場合には−1 を返す。

規　格——*Project LIBC Group*，*XC*

関連項目——chkml，sbrk，sizmem

R

read

用　　途――ファイルから読み込む。

書　　式――#include <unistd.h>

```
int read (int fildes, char *buff, unsigned int n);
```

解　　説――read関数は，*fildes*で指定した**ファイルハンドル**が指すファイルから*n*バイトを読み込み，そのデータを*buff*が指す領域に転送する。ただし，読み込み中に何らかのエラーが発生するかファイル終端に達すると，read関数はその時点で読み込みを中断する。その結果，**ファイルポインタ**は読み込んだバイト数だけ進む。

もし読み込みを開始した時点で，すでにファイルの終端に達していた場合は0を返す。また，*n*が0だった場合は何も行わない。

ファイルが**バイナリモード**でオープンされていた場合，読み込まれたデータはすべてそのまま*buff*に格納される。しかし**テキストモード**でオープンされていた場合は，データ中に含まれるすべてのCRLFをLFに変換してから*buff*に格納する。したがって，格納したデータの長さが*n*に満たない場合がある。

戻り値――正常に読み込めたか途中でファイル終端に達した場合は，実際に読み込んで*buff*に転送したバイト数を返す。何らかのエラーによって失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

*n*が0だったり，読み込む前にすでにファイル終端に達していた場合は，0を返す。

- EBADF　　不正な*fildes*を指定した

注　　意――*buff*は，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――fcntl，lseek，open，write

readdir

用　　途——次のディレクトリストリームの内容を読み込む。

書　　式——

```
#include <dirent.h>
struct dirent *readdir (DIR *dirp);
```

解　　説——readdir 関数は，*dirp*の指すディレクトリストリームの次のエントリへのポインタを返す。dirent 構造体は次のとおり。

```
struct dirent {
    ino_t d_ino;                /* i-node 番号 */
    char d_name[NAME_MAX + 1]; /* ファイルネーム */
};
```

戻 り 値——正常にエントリが読み込めた場合はそのポインタを返し，ディレクトリの終わりに達するなど失敗した場合は NULL を返す。

注　　意——オープンされた後に，そのディレクトリに新規作成されたファイルは対象にはならない。操作対象にするためには再度オープンする必要がある。

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——closedir，readdir，rewinddir，seekdir，telldir

変　　更——従来と現在の *LIBC* では dirent 構造体が非互換である。現在の dirent 構造体は次のとおり。

```
struct dirent {
  ino_t d_ino;               /* i-node 番号 */
  off_t d_size;              /* ファイルサイズ */
  mode_t d_mode;             /* 拡張 UNIX ファイルモード */
  time_t d_time;             /* 最終変更時間 */
  char d_name[NAME_MAX + 1]; /* ファイル名 */
};
```

R

readlink

用　途——シンボリックリンクファイルのリンク先を取得する。

書　式——#include <dos.h>

```
int readlink (const char *linkpath,
              char *namebuf, int buflen);
```

解　説——readlink 関数は，*linkpath* で指定したシンボリックリンクファイルが指しているリンク先ファイル (*linkpath* ファイルの中身) のパス名を求め，*namebuf* の指す *buflen* バイトの領域にコピーする。

戻り値——正常にパスが求められた場合はリンク先のパスの長さ返し，失敗した場合は–1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　指定されたファイルがシンボリックリンクではない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOENT　指定されたファイルが存在しない
- ENOSYS　実体のパスが取得できないか，または **lndrv** が常駐していない
- ERANGE　バッファオーバー

注　意——readlink 関数の動作はすべて **lndrv** に依存するため，**lndrv** 以外のシンボリックリンクドライバでは使用することができない。また，将来 **lndrv** が変更された場合も動作しなくなる可能性がある。

namebuf は，最低でも *buflen* 以上の大きさの領域を指していなければならない。

規　格——*Project LIBC Group*，*AES/OS*，*SYSV*

関連項目——_dos_importlnenv，_dos_lfiles，_dos_symlink，symlink

realloc

用　途――メモリブロックを再確保する。

書　式――#include <stdlib.h>

```
void *realloc (void *ptr, size_t size);
```

解　説――realloc 関数は現在 *ptr* が指しているメモリ領域が，*size* バイトの大きさになるようにメモリブロックのサイズを変更あるいは移動する。

通常，malloc 関数や calloc 関数で確保されたメモリ領域は，指定されたバイト数よりやや大きなサイズをもっていることが多い。そのため，*size* がその余裕で吸収することができたり，現在のサイズよりも小さい場合はメモリ領域は移動せず，サイズだけが変更される。

しかし余裕分だけでは吸収できなかったり，その場でメモリ領域を拡大できないような場合は，realloc 関数は新しい場所に *size* バイト分の領域を確保し，現在のメモリ領域の中身をそっくりコピーすることで対処する。この場合，返されるポインタの値は *ptr* とは異なる。

また，確保されたメモリ領域の先頭アドレスは，いかなるデータ型にでもキャストト可能なように CPU 依存の**アラインメント**に調整される。

戻り値――正常に変更できた場合はその領域へのポインタを返し，失敗した場合には NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなったか，または制限値に達した

R

注　意――もし *size* が現在のメモリ領域のサイズよりも小さい場合，縮小された部分 (メモリ領域の後ろから削られる) の内容は，暗黙のうちに破棄される。

ptr に，malloc 関数や calloc 関数で確保したメモリ領域へのポインタ以外を指定した場合の動作は未定である。

規　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――brk，calloc，free，malloc，rbrk，sbrk

remove

用　途——ファイル/ディレクトリを削除する。

書　式——

```
#include <stdio.h>
int remove (const char *path);
```

解　説——remove 関数は *path* で指定したパス名を削除する。もしパス名が通常ファイルならば remove 関数は unlink 関数を呼び出し，ディレクトリならば rmdir 関数を呼び出す。

戻り値——正常に削除できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOTEMPTY　ディレクトリが空ではない
- ENOENT　指定したパス名が見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- EDEVFS　指定したパス名はデバイスである

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——rmdir，unlink

rename

用　途——ファイル名を変更する。

書　式——#include <stdio.h>

```
#include <stdio.h>
int rename (const char *old, const char *new);
```

解　説——rename 関数は，*old*の指すファイル名を*new*の指すファイル名に変更または移動する。また，パス名がディレクトリを指している場合は，指定のパスにディレクトリを移動する。このとき*new*のディレクトリがすで存在する場合，このディレクトリが空である場合に限り削除して，*old*のディレクトリを移動する。*old*は，既存のファイルまたはディレクトリのパス名でなければならない。

戻り値——正常にファイル名の変更または移動ができた場合は0を返し，失敗した場合は−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ファイルが見つからない
- EXDEV　ファイルシステムが異なる
- EINVAL　*new*のパスに*old*のパスが含まれている
- ENOTDIR　ファイルがすでに存在する
- EISDIR　ディレクトリがすでに存在する
- ENOTEMPTY　*new*ディレクトリが空でない

注　意——ファイルおよびディレクトリの移動は，ファイルシステム(すなわちドライブ)をまたがって行うことはできない。

ディレクトリの移動は疑似的なハードリンクを使用して実現している。

規　格——***Project LIBC Group***，　***ANSI C***，　***POSIX.1***，　***4.3BSD***，　***SYSV***

関連項目——_dos_link，_dos_unlink

R

rewind

用　　途――ファイルストリームのファイルポインタを先頭に戻す。

書　　式――

```
#include <stdio.h>
void rewind (FILE *stream);
```

解　　説――rewind関数は，*stream*で指定されたファイルストリームのファイルポインタを先頭に戻す。また，ストリームに設定されていた終端指示子とエラー指示子がクリアされる。

戻 り 値――特に戻り値はないが，失敗した場合には変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである
- ESPIPE　ファイルストリームに関連するファイルはシークできない
- EINVAL　不正なオフセットを指定した

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――fseek，fsetpos

rewinddir

用　途——ディレクトリストリームの位置を先頭に戻す。

書　式——#include <dirent.h>

```
#include <dirent.h>
int rewinddir (DIR *dirp);
```

解　説——rewinddir 関数は *dirp* が指すディレクトリストリームの位置を，このディレクトリの先頭にリセットする。

戻り値——位置を正常にリセットできた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EPERM　不正なディレクトリストリームを指定した

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——closedir，readdir，rewinddir，seekdir，telldir

rindex

用　　途——文字列中から指定文字が最後に現れる位置を検索する。

書　　式——#include <string.h>

```
char *rindex (const char *string, int character);
```

解　　説——rindex関数は*string*が指す文字列中から，*character*で指定された文字が最後に現れる位置を検索する。*string*末尾のnull文字も検索の対象となりうる。*character*は，内部でint型からchar型に変換される。

戻り値——*string*中で最後に現れる*character*へのポインタを返し，見つからない場合はNULLを返す。

注　　意——1文字は1バイトとなる。

規　　格——***4.3BSD***

関連項目——memchr，strchr，strcspn，strpbrk，strrchr，strspn，strstr，strtok

rmdir

用　途——ディレクトリを削除する。

書　式——#include <sys/stat.h>

```
int rmdir (const char *path);
```

解　説——rmdir 関数は *path* で指定したディレクトリを削除する。

戻り値——正常に削除できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOTEMPTY　ディレクトリが空ではない
- ENOENT　不正なパス名を指定した
- ENOTDIR　ディレクトリではない
- EROFS　リードオンリーファイルシステムである
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——mkdir，remove，unlink

R

SET_FRAME

用　途――フレームポインタに使用するレジスタを指定する。

書　式――#include <interrupt.h>

```
SET_FRAME (int var, reg);
```

解　説――SET_FRAME は *var* に *reg* で指定したレジスタを，関数内部のスタックフレームのフレームポインタに指定し，正しくコンパイルされるようにする。

```
SET_FRAME ("a5");
```

注　意――SET_FRAME は変数の宣言を行うマクロなので，文法的に変数宣言が行える場所にしか記述することができない。

互換性――この機能は *X68000 Programming Series #1 X68000 Develop.* に付属する吉野氏が移植された **GCC** でしか使用することができない。移植性を考慮するならば使用しないこと。

規　格――***Project LIBC Group***

関連項目――PRAMREG，RETREG

_splitpath

用　途——パスを構成要素に分解する。

書　式——#include <stdlib.h>

```
void _splitpath (const char *path, char *drive, char *dir,
                 char *fname, char *ext);
```

解　説——_splitpath 関数は，*path*で指定されたパスをそれぞれの構成要素に分解し，その結果を引数で指定された領域に格納する。それぞれの引数の意味は次のとおりである。

- *drive*　ドライブ名。たとえば“A:”など。もし *path* にドライブ名がなければ空文字列が入る
- *dir*　ディレクトリ名。たとえば“foo/bar/”など。もし *path* にディレクトリ名がなければ空文字列が入る
- *fname*　ファイル名。たとえば“sample”など拡張子を除いた部分。もし *path* にファイル名がなければ空文字列が入る
- *ext*　ファイルの拡張子。たとえば“.c”など。もし *path* に拡張子がなければ空文字列が入る

戻り値——なし。

注　意——_splitpath 関数はパスの構成要素を文字的に分解するだけであり，それぞれの構成要素が有効なものになるかどうかについてはチェックしない。

規　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——_fullentry，_fullpath，_makepath

S

_start

用　途——プログラムの実行開始処理を実行する。

書　式——#include <sys/xstart.h>

```
void __volatile _start (struct _mep *mp);
```

解　説——_start 関数は，プログラムの実行に必要なライブラリやシステムに関するすべての初期化を行い，正しくプログラムをセットアップする。具体的に，_start 関数は以下のような処理を行う。

- **Human68k** から実行された直後，カレントプロセスは割り当て可能なすべてのメモリが与えられている。_start 関数はこのメモリ空間をスタック領域，ヒープ領域などの論理的なブロックに分け，余ったメモリ空間をシステムに返却する
- **Human68k** から渡されたコマンドラインを解析し（HUPAIR エンコードの展開も含む），引数配列を作成する
- プロセス独自の環境変数エリアを作成し，そこに現在の環境をすべてコピーする。以後，環境エリアに対する操作はすべてこのコピー領域に対して行われる
- スタックポインタを設定し，カレントプロセス内部のスタック領域を使用するように手配する
- CPU タイプやコプロセッサの有無を判定し，内部の演算処理を変更するフラグを立てる
- カレントプロセスの実行開始時間を記録し，後で clock 関数によって経過時間が求められるようにする
- システム制限値を取り込み，それを用いてライブラリ内部を初期設定する
- 標準入出力などの最初からオープンされている低水準ファイルハンドルをライブラリ内部に取り込み，正しく処理できるようにする
- 同じく標準入出力などの最初からオープンされている高水準ファイルストリームをライブラリ内部に取り込み，正しく処理できるようにする
- シグナル処理や例外処理，タイマ処理を行うために必要な DOS や IOCS のベクタを書き換え，処理を変更する
- C++のグローバルコンストラクタをプログラム領域から検索して呼び出す

_start 関数は以上のような処理を行った後，メインルーチンである main 関数を呼び出し，ユーザプログラムの実行を開始させる。

戻 り 値――なし。_start 関数は決して戻ってこない。

互 換 性――_start 関数の内部処理は *XC* のそれとはかなり異なっており，既存のライブラリのうちで_main 関数を置き換える必要があるもの，たとえば cshwild，jshwild，**SX–Window** プログラムなどとは共存させることができない。

もしリンクして無理に共存させた場合，その動作およびそれによって引き起こされる結果は不定であり一切保証されない。

規　　格――***Project LIBC Group***

関連項目――_dos_setblock，_dos_super，environ，_iocs_ontime，_main

S

_sysroot

用　途——環境変数 “SYSROOT” を用いてパス名を再構成する。

書　式——#include <sys/xglob.h>

```
char *_sysroot (char *buff, const char *path, size_t n);
```

解　説——_sysroot 関数は，*path* で指定されたパス名と環境変数 SYSROOT から得られたルートディレクトリのパス名を結合し，その結果を *buff* が指す *n* バイトの領域に格納する。ただし，環境変数 SYSROOT が未定義の場合は，単純に *path* をコピーしたものを返す。

たとえば，環境変数 SYSROOT が “A:/root” に設定されていた場合，*path* に “/etc/passwd” を指定すると，結果は次のようになる。

“A:/root/etc/passwd”

ここで_sysroot 関数は，環境変数 SYSROOT の値と *path* を結合し，その結果のパス名が正しくなるように，**パス区切り記号**を加えたり取り除いたりする。

戻り値——正常に処理できた場合は *buff* を返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ERANGE　結果のパス名が *n* バイトより長い

注　意——*buff* は，結果のパス名を格納するだけの十分な大きさの領域を指していなければならない。

_sysroot 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　格——***Project LIBC Group***

関連項目——getenv

sbrk

用　途——ブレーク値を変更する。

書　式——#include <stdlib.h>

```
void *sbrk (int incre);
```

解　説——sbrk 関数は，現在の**ブレーク値**を今よりも *incre* バイトだけ大きくあるいは小さくする。ただしブレーク値は，正確に *incre* バイト分増減するわけではなく，*incre* バイト以上変化したうえで，最も近い**ページ境界** (*LIBC* では疑似的に 4096 バイトをページサイズとする) に整列される。ただし，**ヒープ領域**は決して 0 バイト未満にはならない。

ブレーク値についての詳細な説明は brk 関数を参照のこと。

戻り値——正常にブレーク値を変更できた場合は変更前のブレーク値を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなったか，または制限値に達した

規　格——*Project LIBC Group*，*XC*，*4.3BSD*，*SYSV*

関連項目——brk，chkml，rbrk，sizmem

scanf

用　　途――標準入力ファイルストリームからフォーマット入力を行う。

書　　式――#include <stdio.h>

```
int scanf (const char *format, ...);
```

解　　説――scanf関数は*format*で指定した入力フォーマット文字列にしたがって，必要なデータ列を標準入力に割り当てられたファイルストリームから入力し，その結果を引数の指す領域に格納する。

入力フォーマット文字列の指定方法や詳細な説明については，fscanf関数を参照のこと。

戻り値――正常に入力できた場合は実際に入力できた項目数を返し，失敗した場合はEOFを返して，ストリームのエラー指示子を設定する。失敗した場合は，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　　意――引数に指定するポインタは，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目――fscanf，sscanf

seekdir

用　　途——ディレクトリストリームの位置を変更する。

書　　式——#include <dirent.h>

```
int seekdir (DIR *dirp, long location);
```

解　　説——seekdir 関数は，*dirp* が指すディレクトリストリームの位置を *location* で指定される位置に設定する。

戻 り 値——正常に位置を設定できた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ファイルが見つからない
- EPERM　不正なディレクトリストリームを指定した

規　　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——closedir，readdir，rewinddir，seekdir，telldir

S

setbuf

用　　途――ファイルストリームにバッファを割り当てる。

書　　式――#include <stdio.h>

```
void setbuf (FILE *stream, char *buff);
```

解　　説――setbuf 関数は，*stream*で指定したファイルストリームで使用するバッファ領域を設定する。通常，ファイルストリームをオープンすると自動的にフルバッファリングモードでバッファが確保されるが，setbuf 関数を用いれば，そのバッファをユーザが独自に変更できる。

バッファは*buff*が NULL でなければフルバッファリングモードで，NULL ならばノンバッファリングモードで使用される。

戻り値――なし。

注　　意――*buff*に NULL 以外の値を指定するときは，*buff*は少なくとも BUFSIZ バイトの領域を指していなければならない。なお，BUFSIZ は<stdio.h>に定義されている。

setbuf 関数で指定したバッファ領域は，fclose 関数によってファイルストリームをクローズしても自動的には解放されない。よって，ユーザが責任をもって処理する必要がある。

setbuf 関数を呼び出すと，ストリームをオープンしたときに自動的に割り当てられたバッファ領域は解放される。

setbuf 関数はバッファにデータがない状態で実行する必要がある。つまり，ファイルストリームをオープン/シーク/フラッシュした直後に実行しなくてはならない。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――fclose，fopen，setvbuf

setgid

用　途——グループ ID を変更する。

書　式——

```
#include <unistd.h>
int setgid (gid_t gid);
```

解　説——setgid 関数は，カレントプロセスの実グループ ID や実効グループ ID を *gid* に変更する。

戻り値——正常に変更できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な *gid* を指定した

互換性——**Human68k** にはユーザ ID やグループ ID の概念がないので，setgid 関数がこれらの値に影響されることはない。また，*LIBC* 内ではデフォルトの仮想ユーザ ID および仮想グループ ID を root と仮定しているため，setgid 関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——getgid，setuid

S

setgrent

用　途——グループファイルへのアクセスをファイル先頭に戻す。

書　式——#include <grp.h>

```
void setgrent (void);
```

解　説——setgrent関数は，getgrent関数などでアクセス中のグループファイルのファイルポインタを先頭に戻す。getgrnam関数やgetgrgid関数のインタフェイスを用いずにグループを検索する場合には，明示的にendgrent関数によってアクセスを終了すること。

戻り値——なし。

規　格——*Project LIBC Group*，*4.3BSD*，*SYSV*

関連項目——endgrent，getgrent，getgrgid，getgrnam

setjmp

用　途——大域ジャンプ用のジャンプポイントを設定する。

書　式——#include <setjmp.h>

```
int setjmp (jmp_buf env);
```

解　説——setjmp 関数は呼び出された時点（ジャンプポイント）での環境を，jmp_buf 型の配列 *env* に記憶する。この環境 *env* は，後で longjmp 関数に引数として与えると制御をジャンプポイントに移すことができる。

戻り値——記憶を行った場合 setjmp 関数は 0 を返すが，longjmp 関数によって制御が移された場合は，longjmp 関数で指定した *rval* が戻り値となる。この *rval* は必ず 0 以外の値をとるので，両者は区別することができる。

注　意——longjmp 関数によって正しく環境を復帰させるためには，setjmp 関数は以下に述べる文脈にしか存在してはならない。それ以外の場合の動作は一切保証されない。

- 選択文（if，switch）か反復文（while，for，do while）の制御式
- 制御式中で，一方が汎整数定数式である関係演算子など非等号演算子のオペランド
- 制御式中の！演算子のオペランド
- 式中のすべての式。void 型も可

互換性——*ANSI C* では setjmp 関数はマクロとして実装されることが定められているが，*LIBC* では関数として定義している。忠実に *ANSI C* に合致させようとするならば，_setjmp 関数を作成し，setjmp は_setjmp 関数への別名定義にすればよい。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——longjmp，siglongjmp，sigsetjmp

setmode

用　　途――ファイルの変換モードを変更する。

書　　式――#include <unistd.h>

```
int setmode (int fildes, int mode);
```

解　　説――setmode 関数は，*fildes*で指定したファイルハンドルのテキスト/バイナリ変換モードを *mode*に変更する。*mode*には次の値を指定することができる。

- O_TEXT　　データ変換モードをテキストモードに変更する
- O_BINARY　データ変換モードをバイナリモードに変更する

戻 り 値――正常に変更できた場合は変更前の変換モードを返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な *mode*を指定した
- EBADF　不正な *fildes*を指定した

規　　格――*Project LIBC Group*，　*MS-C 7.0*

関連項目――creat，fmode，fopen，open

setpgid

用　途——プロセスグループIDを変更する。

書　式——#include <unistd.h>

```
int setpgid (pid_t pid, gid_t pgid);
```

解　説——setpgid関数は，*pid*で指定したプロセスのプロセスグループIDを*pgid*に変更する。

戻り値——正常に変更できた場合は0を返し，失敗した場合には-1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　不正な*pgid*を指定した
- ESRCH　不正な*pid*を指定した

注　意——setpgid関数は，カレントプロセス以外のプロセスに対しては実行できない。すなわち*pid*には，必ずgetpid関数で求めた値を指定する必要がある。

互換性——**Human68k**にはプロセスIDやユーザID，グループIDの概念がないので，setpgid関数がこれらの値に影響されることはない。また，*LIBC*内ではデフォルトの仮想ユーザIDおよび仮想グループIDをrootと仮定しているため，setpgid関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——getpgrp，getpid，setsid

S

setpwent

用　　途——パスワードファイルへのアクセスをファイル先頭に戻す。

書　　式——#include <pwd.h>

```
void setpwent (void);
```

解　　説——setpwent 関数は，getpwent 関数などでアクセス中のパスワードファイルのファイルポインタを先頭に戻す。

getpwnam 関数や getpwuid 関数のインタフェイスを用いずにユーザを検索する場合には，明示的に endpwent 関数によってアクセスを終了すること。

戻 り 値——なし。

規　　格——*Project LIBC Group*，　*4.3BSD*，　*SYSV*

関連項目——endpwent，getpwent，getpwnam，getpwuid

setrlimit

用　途——システム制限値を再設定する。

書　式——#include <sys/resources.h>

```
int setrlimit (int resource, struct rlimit *rlp);
```

解　説——setrlimit 関数は現在の環境における *resource* で指定したシステム制限値を，*rlp* で指定した値に再設定する。*resource* には次の値を指定することができ，再設定は以下のようにして行われる。

- RLIMIT_CPU　　CPU 消費時間は再設定不可
- RLIMIT_FSIZE　　最大ファイルサイズは再設定不可
- RLIMIT_DATA　　ヒープ領域の最大サイズは正でかつ最大値以下ならば，再設定可能
- RLIMIT_STACK　　スタック領域の最大サイズは再設定不可
- RLIMIT_CORE　　コアファイルの最大サイズは正でかつ最大値以下ならば，再設定可能
- RLIMIT_RSS　　メモリ空間の最大サイズは再設定不可
- RLIMIT_NOFILE　　最大オープンファイル数は再設定不可

また，rlimit 構造体は次のとおり。

```
struct rlimit {
    int rlim_cur; /* 現在の制限値 */
    int rlim_max; /* 設定できる最大の制限値 */
};
```

戻り値——正常に設定できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EPERM　　値を再設定することができない
- EINVAL　　不正な *resource* を指定した

規　格——***Project LIBC Group*，　*4.3BSD***

関連項目——getrlimit

S

setsid

用　　途――新しいプロセスグループIDを形成する。

書　　式――#include <unistd.h>

```
int setsid (void);
```

解　　説――setsid関数はカレントプロセスを，プロセスグループのリーダーとした新しいセッションを形成する。

戻 り 値――正常に変更できた場合は0を返し，失敗した場合には–1を返す。

互 換 性――**Human68k**にはプロセスIDやユーザID，グループIDの概念がないので，setsid関数がこれらの値に影響されることはない。また，*LIBC*ではデフォルトの仮想ユーザIDおよび仮想グループIDをrootと仮定しているため，setsid関数は何ら影響をおよぼさないとはいえ，変更は必ず成功する。

規　　格――***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――setpgid

setuid

用　　途——ユーザIDを変更する。

書　　式——#include <unistd.h>

```
int setuid (uid_t uid);
```

解　　説——setuid関数は，カレントプロセスの実ユーザIDおよび実効ユーザIDを*uid*に変更する。

戻 り 値——正常に変更できた場合は0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　不正な*uid*を指定した

互 換 性——**Human68k**にはユーザIDやグループIDの概念がないので，setuid関数がこれらの値に影響されることはない。また，*LIBC*ではデフォルトの仮想ユーザIDおよび仮想グループIDをrootと仮定しているため，setuid関数は何ら影響をおよぼさない。とはいえ，変更は必ず成功する。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——geteuid，getuid，setgid

setvbuf

用　途――ファイルストリームにバッファを割り当てる。

書　式――

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buff, int type, size_t size);
```

解　説――setvbuf関数は，*stream*で指定したファイルストリームで使用するバッファ領域を設定する。以後，ファイルストリームは*buff*が指す*size*バイトの領域を，*type*で指定したバッファリング方式で使用する。ただし，*buff*にNULLが指定された場合は，setvbuf関数が自動的に*size*バイトの領域を割り当てる。

*type*には以下のマクロ値を指定する。

- _IOFBF　フルバッファリング方式
- _IOLBF　ラインバッファリング方式
- _IONBF　ノンバッファリング方式

ストリームのバッファリング方式がフルバッファリングの場合，すべての入出力は*size*バイト分メモリ内に蓄えられ，バッファがいっぱいになった時点で実際に入出力が行われる。

ラインバッファリングの場合，これに加えて改行文字に出会ったときに入出力が行われる。また出力の場合，ノンバッファリング/行バッファリングされているストリーム(ほかのストリームも対象になる)に対して，入力要求が行われたときにも出力が行われる。

一方ノンバッファリングの場合，すべての入出力はその場で行われ，メモリ内で蓄えられることも遅延されることもない。

戻り値――正常に割り当てることができた場合は0を返し，失敗した場合には0以外の値を返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- EINVAL　不正な*type*を指定した

注　意――*buff*にNULL以外の値を指定するときは，*buff*は少なくとも*size*バイトの領域を指していなければならない。

setvbuf関数で指定したバッファ領域は，fclose関数によってファイルストリームをクローズしても自動的には解放されない。よってユーザが責任をもって処理する必要がある。

setvbuf 関数を呼び出すと，ストリームをオープンしたときに自動的に割り当てられたバッファ領域は解放される。

setvbuf 関数はバッファにデータがない状態で実行する必要がある。つまり，ファイルストリームをオープン/シーク/フラッシュした直後に実行しなくてはならない。

規　格――*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――fclose，fopen，setbuf

S

sigaction

用　　途――シグナル発生時の動作を再設定または取得する。

書　　式――#include <signal.h>

```
int sigaction (int sig, const struct sigaction *act,
                 struct sigaction *oact);
```

解　　説――sigaction 関数は *sig* で指定した番号のシグナルの**シグナル動作**を，もし *act* が NULL でなければそれの値で再設定し，元の設定情報を *oact* が NULL でなければその領域へ格納する。sigaction 構造体は次のとおり。

```
struct sigaction {
    void (*sa_handler) (int); /* ハンドへのポインタ */
    sigset_t sa_mask;         /* シグナル発生時のマスク */
    int sa_flags;             /* シグナルフラグ */
};
```

また，上記の各メンバの意味と値の設定方法は次のとおり。

- sa_handler　**シグナルハンドラ**へのポインタ。signal 関数で設定するハンドラと同じ
- sa_mask　シグナルが発生すると，ライブラリはまずシグナルハンドラが多重に呼び出されないように，当該シグナルをマスクする。そのとき，ここで指定されたシグナルマスクも同時に適用される。この値を設定することで，当該シグナルが発生したときに任意のシグナルマスクを加えることができる
- sa_flags　シグナル発生時の処理の仕方を制御する。もしここに SA_RESETHAND が設定されている場合は，シグナルハンドラが呼び出されるたびにシグナルハンドラ sa_handler は，デフォルトの設定 (SIG_DFL) に戻される。一般に ***4.3BSD*** ではシグナルハンドラはデフォルトに戻らず，***SYSV*** では元に戻る

シグナル動作に関する詳細な説明は，signal 関数を参照のこと。

戻 り 値――正常に設定できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な *sig* を指定したか，または SIGKILL

規　格——*POSIX.1*, *XPG3*, *AES/OS*, *SYSV*

関連項目——kill, raise, sigaddset, sigblock, sigdelset, sigemptyset, sigfillset, sigismember, signal, sigprocmask, sigsuspend

S

sigaddset

用　　途——シグナルセットへの追加を実行する。

書　　式——#include <signal.h>

```
int sigaddset (sigset_t *set, int signo);
```

解　　説——sigaddset 関数は *set* が指す領域に格納されているシグナルセットに，*signo* で指定したシグナルを追加する。

戻 り 値——正常に追加できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正なシグナル *signo* を指定した

注　　意——*set* が指すシグナルセットは，少なくとも 1 回は sigemptyset 関数あるいは sigfillset 関数によって初期化されている必要がある。

通常，sigaddset 関数はマクロとして定義されるが，__NO_SIGNAL_INLINE__ が定義された場合には実体をもつ関数となる。なおマクロ版の sigaddset 関数は，エラーチェックを行わないので注意すること。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——sigaction，sigdelset，sigemptyset，sigfillset，sigismember，sigpending，sigprocmask，sigsuspend

sigblock

用　途——シグナルをブロックする。

書　式——#include <signal.h>

```
int sigblock (int mask);
```

解　説——sigblock 関数は *mask* で指定されたシグナルマスクを，カレントシグナルマスクとし，当該シグナルが配信されるのを抑制する。ブロックされたシグナルが発生すると，そのシグナルはペンディングシグナルセットに追加され，マスクが適切な値になったときに(解除されたときに)，あらためて配信される。

戻り値——追加される前のシグナルマスクを返す。

注　意——sigblock 関数は sigprocmask 関数の古い形のインタフェイスである。したがって戻り値は sigset_t 型ではなく，int 型となっている。*LIBC* では sigset_t 型は int 型と同じであるから問題ないが，移植性を考慮するならば，sigprocmask 関数のインタフェイスを用いるべきである。

規　格——*Project LIBC Group*，　*4.3BSD*

関連項目——kill, raise, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember, signal, sigprocmask, sigsuspend

S

sigdelset

用　　途——シグナルセットからの削除を実行する。

書　　式——#include <signal.h>

```
int sigdelset (sigset_t *set, int signo);
```

解　　説——sigdelset 関数は *set* が指す領域に格納されているシグナルセットから，*signo* で指定したシグナルを削除する。

戻 り 値——正常に削除できた場合は0を返し，失敗した場合には−1を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正なシグナル *signo* を指定した

注　　意——*set* が指すシグナルセットは，少なくとも1回は sigemptyset 関数あるいは sigfillset 関数によって初期化されている必要がある。

通常，sigdelset 関数はマクロとして定義されるが，__NO_SIGNAL_INLINE__ が定義された場合には実体をもつ関数となる。なおマクロ版の sigdelset 関数は，エラーチェックを行わないので注意すること。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——sigaction，sigaddset，sigemptyset，sigfillset，sigismember，sigpending，sigprocmask，sigsuspend

sigemptyset

用　　途——シグナルセットの初期化を実行する。

書　　式——#include <signal.h>

```
int sigemptyset (sigset_t *set);
```

解　　説——sigemptyset 関数は，*set* が指す領域に格納されているシグナルセットを初期化する。sigaddset 関数や sigdelset 関数などの操作を行う前には，必ず sigemptyset 関数や sigfillset 関数によって初期化されている必要がある。

戻 り 値——正常に初期化できた場合は 0 を返し，失敗した場合には−1 を返す。

注　　意——通常，sigemptyset 関数はマクロとして定義されるが，__NO_SIGNAL_INLINE__が定義された場合には実体をもつ関数となる。なおマクロ版の sigemptyset 関数は，エラーチェックを行わないので注意すること。

規　　格——*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——sigaction，sigaddset，sigdelset，sigfillset，sigismember，sigpending，sigprocmask，sigsuspend

S

sigfillset

用　　途――シグナルセットを初期化し，すべてのシグナルを設定する。

書　　式――

```
#include <signal.h>
int sigfillset (sigset_t *set);
```

解　　説――sigfillset 関数は *set* が指す領域に格納されているシグナルセットを初期化し，定義されているすべてのシグナルをシグナルセットに追加する。sigaddset 関数や sigdelset 関数などの操作を行う前には，必ず sigemptyset 関数 や sigfillset 関数によって初期化されている必要がある。

戻 り 値――正常に初期化できた場合は0を返し，失敗した場合には−1を返す。

注　　意――通常，sigfillset 関数はマクロとして定義されるが，__NO_SIGNAL_INLINE__が定義された場合には実体をもつ関数となる。なおマクロ版の sigfillset 関数は，エラーチェックを行わないので注意すること。

規　　格――***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目――sigaction, sigaddset, sigdelset, sigemptyset, sigismember, sigpending, sigprocmask, sigsuspend

sigismember

用　途——シグナルセットに指定したシグナルが設定されているかどうかを調べる。

書　式——#include <signal.h>

```
int sigismember (sigset_t *set, int signo);
```

解　説——sigismember 関数は *set* が指す領域に格納されているシグナルセットに，*signo* で指定したシグナルが設定されているかどうかを調べる。

戻り値——シグナルが設定されていれば 0 以外の値を返し，設定されていなければ 0 を返す。またチェックに失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正なシグナル *signo* を指定した

注　意——*set* が指すシグナルセットは，少なくとも 1 回は sigemptyset 関数あるいは sigfillset 関数によって初期化されている必要がある。

通常，sigismember 関数はマクロとして定義されるが，__NO_SIGNAL_INLINE__ が定義された場合には実体をもつ関数となる。なおマクロ版の sigismember 関数は，エラーチェックを行わないので注意すること。

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——sigaction，sigaddset，sigdelset，sigemptyset，sigfillset，sigpending，sigprocmask，sigsuspend

S

siglongjmp

用　途——大域ジャンプを実行する。

書　式——

```
#include <setjmp.h>
void __volatile siglongjmp (sigjmp_buf env, int rval);
```

解　説——siglongjmp関数は，sigsetjmp関数でsigjmp_buf型の配列*env*に記憶したジャンプポイントに大域ジャンプを行い，その結果，プログラムカウンタは*env*に記憶されたジャンプポイントに移る。一方ジャンプポイントでは，記憶を行ったsigsetjmp関数が見かけ上の戻り値として，*rval*を返すように見える。

また siglongjmp関数は，*env*が sigsetjmp関数によってカレントシグナルマスクが記憶されている場合に限り，シグナルマスクを*env*から復帰させる。

siglongjmp関数は，割り込みルーチンから正常に復帰するために用いる。

戻り値——なし。この関数は決して戻ってこず，*env*で指定したジャンプポイントに実行を移す。

注　意——sigsetjmp関数での戻り値*rval*は，通常の場合と siglongjmp関数された場合とを区別しなくてはいけないので，0以外の値でなくてはならない。*rval*に0を設定した場合は1とみなされる。

*env*は，あらかじめ sigsetjmp関数によって環境を格納しておく必要がある。もし格納されていなかった場合，その動作は一切保証されない。

siglongjmp関数が実行され，ジャンプポイントの位置に戻った場合，レジスタに割り当てられたauto型変数を除くアクセス可能なすべてのオブジェクトは，siglongjmp関数が実行されたときの値をもつ。

sigsetjmp関数を実行した関数がreturnやその他の大域ジャンプなどによって，より上位 (呼び出し元) の関数に戻ってしまった場合，その後でsiglongjmp関数によってジャンプポイントに戻ろうとしても，その動作は保証されない。

シグナル処理中に発生した別のシグナル処理の最中にsiglongjmp関数を実行した場合，すなわち入れ子になったシグナル割り込み処理からのsiglongjmp関数の動作は保証されない。

規　格——***ANSI C，POSIX.1，XPG3，AES/OS，4.3BSD，SYSV***

関連項目——longjmp，setjmp，sigaction，sigsetjmp

signal

用　途——シグナルハンドラを設定または登録する。

書　式——#include <signal.h>

```
void (*signal (int sig, void (*func) (int))) (int);
```

解　説——signal 関数は，*sig* で指定したシグナルに対するシグナルハンドラとして *func* を登録する。

sig で指定することができるシグナルとそのデフォルトの処理は，次のとおりである。なお S は強制終了，C はコアダンプ，N はプロセスの中断を表している。また，DOS 例外とは **Human68k** が生成する例外，ソフト例外は *LIBC* が生成する例外，ハード例外はハードウェアが生成する例外を表している。

SIGABRT	S	ソフト例外	ABORT シグナル
SIGINT	S	DOS 例外	コントロール C インタラプト
SIGILL	C	ハード例外	不正な命令を実行した
SIGFPE	C	ハード例外	0 による除算
SIGKILL	S	ソフト例外	KILL シグナル
SIGBUS	C	ハード例外	アドレスエラー
SIGSEGV	C	ハード例外	バスエラー
SIGALRM	S	ハード例外	リアルタイムクロック割り込み
SIGTERM	S	ソフト例外	TERMINATE シグナル
SIGEMT	C	ハード例外	未定義トラップ
SIGSTOP	N	ソフト例外	STOP シグナル
SIGUSR1	S	ソフト例外	USR1 シグナル
SIGUSR2	S	ソフト例外	USR2 シグナル

func はシグナルハンドラの関数へのポインタでなければならないが，それとは別に以下の特殊な値を指定することができる。ただし，いずれの場合にも SIGKILL シグナルに対しては，いかなる値も指定することができない。

- SIG_IGN　シグナルを無視するようにする。ただし，ハードウェア例外など無視しても，結果的にプログラムが暴走してしまうような場合もあるので注意されたい
- SIG_DFL　シグナルハンドラをデフォルトのハンドラに戻す。シグナルはそれぞれにデフォルトのシグナルハンドラをもっており，その処理を行わせるようにする。詳細については次ページで述べる。何も設定されていない場合，すべてのシグナルはデフォルトにこれが指定されている。
- SIG_DOS　シグナル機構は，*LIBC* では例外処理を用いて実装されているが，そのために **Human68k** の例外処理ベクタがフックされている。

S

これを指定することで，指定したシグナルに対する処理をライブラリではなく，**Human68k** のオリジナルの処理に戻すことができる。

シグナルが発生すると，ライブラリはまず発生したシグナルに対して**シグナルマスク**をかけて，シグナルハンドラ実行中，同じシグナルをブロックし，多重にハンドラが呼び出されることがないようにする。次に，そのシグナルに対するハンドラが登録されている場合は，シグナルハンドラを起動する。シグナルハンドラの引数は1つで，発生したシグナルの番号が`int`型の引数で与えられる。シグナルハンドラから制御が戻ると，ライブラリはさきほど設定したマスクを解除し，ペンディングされていたシグナルを解放する。

シグナルハンドラは，`return`文を実行して**割り込み発生ポイント**に復帰するか，`siglongjmp`関数によって別の位置へ**大域ジャンプ**するか，`exit`関数，`abort`関数などの関数を用いてプログラムを終了させるかの選択ができる。しかし，ハードウェア例外から発生するシグナルのハンドラについては注意が必要である。仮に`return`文で元に戻ったとしても，正常に実行を継続できるかどうかは未定義である。

戻り値——正常にハンドラを登録できた場合は前のハンドラへのポインタを返し，失敗した場合には`SIG_ERR`を返して，変数`errno`にその原因を示すエラーコードを設定する。

- `EINVAL`　不正な*sig*を指定したか，または`SIGKILL`だった

注　意——シグナルハンドラは**リエントラント**でない関数を呼び出してはならない。リエントラントな関数とは，実行状態によらずつねに一定の結果を出すことができる関数のことで，たとえば静的領域やヒープ領域にアクセスするなど，外部との接点があるような関数はこれに該当しない。もしリエントラントでない関数を呼び出した場合，その動作は未定義である。

*LIBC*ではある程度強引な方法でシグナル機構を実装しているため，ハードウェア例外に対して，強制終了以外の選択肢をとることは避けるべきである。最悪の場合，リセットする以外に回復する方法がなかったり，副作用でデータの破壊を引き起こすことがある。

規　格——***Project LIBC Group***，***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——`abort`，`exit`，`longjmp`，`sigaction`

sigpending

用　　途——現在ペンディング状態にあるシグナルのセットを取得する。

書　　式——`#include <signal.h>`

```
int sigpending (sigset_t *set);
```

解　　説——sigpending 関数は現在シグナルの配信がブロックされ，ペンディング状態にあるシグナルのシグナルセットを求め，その結果を *set* が指す領域に格納する。

戻 り 値——正常に取得できた場合は 0 を返し，失敗した場合には−1 を返す。

注　　意——*set* が指すシグナルセットは，少なくとも 1 回は sigemptyset 関数あるいは sigfillset 関数によって初期化されている必要がある。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——sigaction, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember, sigprocmask，sigsuspend

S

sigprocmask

用　途——ブロックシグナルセットを取得または変更する。

書　式——#include <signal.h>

```
int sigprocmask (int how, sigset_t *set, sigset_t* oset);
```

解　説——sigprocmask関数は，現在ブロックされているシグナルのセットを取得，再設定する。もし*set*がNULLでなければ，**カレントシグナルマスク**は*how*が示す方法で*set*から再設定され，*oset*がNULLでなければ，元の**ブロックシグナルセット**が*oset*が指す領域に格納される。

*how*には次の値を指定することができる。

- SIG_BLOCK　新しいカレントシグナルマスクは，現在のシグナルマスクと *set* が指すシグナルマスクを加えたものとなる
- SIG_UNBLOCK　新しいカレントシグナルマスクは，現在のシグナルマスクから *set* が指すシグナルマスクを除いたものとなる
- SIG_SETMASK　新しいカレントシグナルマスクは，*set*が指すシグナルマスクで完全に置き換えられる

新しいシグナルマスクが計算され，その結果として，ブロックされていたシグナルが解放されるとそれまでブロックされて**シグナルペンディングセット**に保存されていたシグナルは実際に配信される。

戻り値——正常に取得/設定できた場合は0を返し，失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　不正な *how*を指定した

注　意——*set*が指すシグナルセットは，少なくとも1回はsigemptyset関数あるいはsigfillset関数によって初期化されている必要がある。

規　格——***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——kill，raise，sigaction，sigaddset，sigblock，sigdelset，sigemptyset，sigfillset，sigismember，signal，sigsuspend

sigsetjmp

用　途——大域ジャンプ用のジャンプポイントを設定する。

書　式——#include <setjmp.h>

```
int sigsetjmp (sigjmp_buf env, int savemask);
```

解　説——sigsetjmp 関数は呼び出された時点（ジャンプポイント）での環境を，sigjmp_buf 型の配列 *env* に記憶する。この環境 *env* を，後で siglongjmp 関数に引数として与えてやると，制御をジャンプポイントに移すことができる。

sigsetjmp 関数は setjmp 関数の機能に加え，*savemask* が 0 でなければプロセスのカレントシグナルマスクを記憶する。

戻り値——記憶を行った場合 sigsetjmp 関数は 0 を返すが，siglongjmp 関数によって制御が移された場合は，siglongjmp 関数で指定した *rval* が戻り値となる。この *rval* は必ず 0 以外の値をとるので，両者は区別することができる。

注　意——siglongjmp 関数によって正しく環境を復帰させるためには，sigsetjmp 関数は以下に述べる文脈にしか存在してはならない。それ以外の場合の動作は一切保証されない。

- 選択文（if, switch）か反復文（while, for, do while）の制御式
- 制御式中で，一方が汎整数定数式である関係演算子など非等号演算子のオペランド
- 制御式中の！演算子のオペランド
- 式中のすべての式。void 型も可

規　格——*POSIX.1*, *XPG3*, *AES/OS*, *SYSV*

関連項目——longjmp, setjmp, siglongjmp

S

sigsuspend

用　　途――シグナルが配信されるのを待つ。

書　　式――

```
#include <signal.h>
int sigsuspend (sigset_t *sigmask);
```

解　　説――sigsuspend 関数はカレントシグナルマスクを*sigmask*が指すマスクで置き換え，シグナルが配信されるまで (すなわち sigaction 関数や signal 関数で設定したシグナルハンドラが実行されるか，プロセス自体が終了するまで)，現在実行中のプロセスを中断する。

シグナルが配信され，プロセスの実行が再開すると，カレントシグナルマスクは sigsuspend 関数が実行される以前の値に戻される。

戻 り 値――正常に取得/設定できた場合は 0 を返し，失敗した場合には−1 を返す。

注　　意――*sigmask*が指すシグナルマスクは，少なくとも 1 回は sigemptyset 関数あるいは sigfillset 関数によって初期化されている必要がある。

規　　格――***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目――kill，raise，sigaction，sigaddset，sigblock，sigdelset，sigemptyset，sigfillset，sigismember，signal，sigprocmask

sin

用　途——正弦 $sin(x)$ を求める。

書　式——

```
#include <math.h>
double sin (double x);
#include <sys/xmath.h>
double _f_sin (double x);
double _fe_sin (double x);
double _fpu_sin (double x);
```

解　説——sin 関数は x (単位：ラジアン) の正弦 (サイン) を求める。各関数はそれぞれ次のように動作する。

- sin　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_sin　数値演算コプロセッサ命令を直接ドライブする
- _fe_sin　FLOAT パッケージを呼び出す
- _fpu_sin　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，sin 関数は x の正弦を返す。もし x が非数 (NaN) ならば変数 errno に EDOM を設定して同じく非数を返し，それ以外のエラーでは，変数 errno に ERANGE を設定して計算結果を返す。ただし計算結果は不正確な値になる。

- EDOM　x の値が非数，または±HUGE_VAL
- ERANGE　x の値が大きく計算結果の有効桁が一部失われる場合，または x の値が非つねに大きくて計算結果の有効桁が完全に失われる場合

S

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは sin は _fe_sin の別名となり，__DIRECT_IOFPU__が定義されているときは sin は _fpu_sin の別名となる。また，__DIRECT_FPU__が定義されているときは sin は _f_sin の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　　格――*Project LIBC Group*, *ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目――asin, isnan

sinh

用　途──双曲正弦 $sinh(x)$ を求める。

書　式──

```
#include <math.h>
double sinh (double x);
#include <sys/xmath.h>
double _f_sinh (double x);
double _fe_sinh (double x);
double _fpu_sinh (double x);
```

解　説──sinh 関数は x の双曲正弦 (ハイパボリックサイン) を求める。各関数はそれぞれ次のように動作する。

- sinh　　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_sinh　　数値演算コプロセッサ命令を直接ドライブする
- _fe_sinh　　FLOAT パッケージを呼び出す
- _fpu_sinh　　I/O 数値演算コプロセッサを直接ドライブする

戻り値──正しく値が求められた場合，sinh 関数は x の双曲正弦を返す。もし x が非数 (NaN) ならば変数 errno に EDOM を設定して同じく非数を返し，それ以外のエラーでは変数 errno にその原因となるエラーコードを設定して無限大を返す。

- EDOM　　x の値が非数
- ERANGE　　計算結果がオーバーフローした

S

注　意──<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは sinh は _fe_sinh の別名となり，__DIRECT_IOFPU__が定義されているときは sinh は _fpu_sinh の別名となる。また，__DIRECT_FPU__が定義されているときは sinh は _f_sinh の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　格──*Project LIBC Group，ANSI C，POSIX.1，XPG3，AES/OS，4.3BSD，SYSV*

関連項目──asin，sin

sizmem

用　　途——空きメモリ容量をロングワード単位で調べる。

書　　式——#include <stdlib.h>

```
size_t sizmem (void);
```

解　　説——sizmem 関数は空きメモリ容量，正確には連続して確保できる最大の空きメモリブロックのサイズを調べ，その結果をロングワード単位で返す。

戻 り 値——空きメモリ容量をロングワード (4バイト) 単位で返す。

注　　意——正確には空きメモリ容量とは異なる。また，ここでいう空きメモリブロックとは，**Human68k** が管理しているメモリブロックのことであり，プロセスのヒープ領域から確保されるメモリブロック (管理するのはプロセス自身) ではない。

規　　格——*Project LIBC Group*，　*XC*

関連項目——brk，chkml

sleep

用　途――秒単位のスリープを実行する。

書　式――#include <unistd.h>

```
unsigned int sleep (unsigned int seconds);
```

解　説――sleep 関数は *seconds* で指定した秒数だけ，現在実行中のプロセスをスリープさせる。プロセスは指定した時間スリープするか，シグナルが通知されてシグナルハンドラが呼び出された場合に，スリープ状態から復帰する。

戻り値――指定した時間が経過した場合は 0 を，途中でシグナルによって中断された場合は残りスリープ時間 (秒単位) を返す。

規　格――***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――pause，signal，usleep

spawnl

用　　途——プログラムを実行する。

書　　式——#include <unistd.h>

```
int spawnl (int mode, const char *path, const char *arg0, ...);
```

解　　説——spawnl関数は現在実行中のプログラムに代わり，*path*で指定したプログラムを*mode*の実行モードで起動し，制御を移す。*mode*に指定できる実行モードは次のとおりである。

- P_OVERLAY　execl関数の実行と同じ。詳細はexecl関数を参照のこと
- P_WAIT　子プロセスが終了するまで待機する

spawnl関数には*arg*以降に，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。Cのプログラムは起動時されるとmain関数に*argc*と*argv*という引数配列が渡されるが，*arg*0からの一連の引数は，この*argv*配列を直接指定しているのと同じである。ただし*argv*[0]はプログラム自身を表すので，普通は*path*と同じ値を設定する。また引数列の最後に，終端記号としてNULLを指定すること。

戻 り 値——実行モードがP_OVERLAYならば，正常に実行できた場合，spawnl関数は戻ってこない。失敗した場合は−1を返して，変数errnoにその原因を示すエラーコードを設定する。また実行モードがP_WAITならば，正常に実行できた場合，spawnl関数は子プロセスの終了コードを返す。失敗した場合は−1を返して，変数errnoにその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path*で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　　意——実行しようとするプログラムが**HUPAIR**仕様に準拠したプログラムならば，引数は**HUPAIR**仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR**仕様に準拠していないプログラムの場合，**Human68k**で規定されている通常の方法によって引数が渡される。

規　　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——spawnle，spawnlp，spawnv，spawnve，spawnvp

spawnle

用　途——プログラムを実行する。

書　式——
```
#include <unistd.h>
int spawnle (int mode, const char *path, const char *arg0, ...
             char *const envp[]);
```

解　説——spawnle 関数は現在実行中のプログラムに代わり，*path* で指定したプログラムを *mode* の実行モードで起動し，制御を移す。*mode* に指定できる実行モードは次のとおりである。

- P_OVERLAY　execle 関数の実行と同じ。詳細は execle 関数を参照のこと
- P_WAIT　子プロセスが終了するまで待機する

spawnle 関数には *arg* 以降に，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動時されると main 関数に *argc* と *argv* という引数配列が渡されるが，*arg*0 からの一連の引数は，この *argv* 配列を直接指定しているのと同じである。ただし，*argv* [0] はプログラム自身を表すので，普通は *path* と同じ値を設定する。また引数列の最後に，終端記号として NULL を指定すること。

spawnle 関数は実行するプログラムに与える環境エリアを，*envp* の配列で指定することができる (*envp* は引数列の最後に設定される NULL の次である)。実行されるプログラムが，もし getenv 関数などによって環境変数を検索しようとすると，この配列の内容が検索されることになる。この配列の構造については変数 environ を参照のこと。

戻り値——実行モードが P_OVERLAY ならば，正常に実行できた場合，spawnle 関数は戻ってこない。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。また実行モードが P_WAIT ならば，正常に実行できた場合，spawnle 関数は子プロセスの終了コードを返す。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　　意――実行しようとするプログラムが**HUPAIR**仕様に準拠したプログラムならば，引数は**HUPAIR**仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR**仕様に準拠していないプログラムの場合，**Human68k**で規定されている通常の方法によって引数が渡される。

規　　格――*Project LIBC Group*，　*MS–C 7.0*

関連項目――spawnl，spawnlp，spawnv，spawnve，spawnvp

spawnlp

用　途——プログラムを実行する。

書　式——#include <unistd.h>

```
int spawnlp (int mode, const char *file, const char *arg0, ...);
```

解　説——spawnlp 関数は現在実行中のプログラムに代わり，*file*で指定したプログラムを*mode*の実行モードで起動し，制御を移す。*mode*に指定できる実行モードは次のとおりである。

- P_OVERLAY　exec 関数の実行と同じ。詳細は execlp 関数を参照のこと
- P_WAIT　子プロセスが終了するまで待機する

spawnlp 関数には *arg*以降に，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc*と *argv*という引数配列が渡されるが，*arg*0 からの一連の引数は，この *argv*配列を直接指定しているのと同じである。ただし，*argv*[0] はプログラム自身を表すので，普通は *file*と同じ値を設定する。また引数列の最後に，終端記号として NULL を指定すること。

呼び出されるプログラムは *file*で指定する。もしも *file*が "/" や "\" のようなパスの区切り記号を含んでいれば，*file*はプログラムのパスを示しているとみなされる。しかし区切り記号が含まれていなければ，spawnlp 関数は *file*という名前のファイルを環境変数 path に設定されているディレクトリ中から検索し，最初に見つかったものを実行する。

戻り値——実行モードが P_OVERLAY ならば，正常に実行できた場合 spawnlp 関数は戻ってこない。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。また実行モードが P_WAIT ならば，正常に実行できた場合 spawnlp 関数は子プロセスの終了コードを返す。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*file*で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆

S

に，HUPAIR仕様に準拠していないプログラムの場合，Human68kで規定されている通常の方法によって引数が渡される。

規　　格――*Project LIBC Group*，　*MS–C 7.0*

関連項目――spawnl，spawnle，spawnv，spawnve，spawnvp

spawnv

用　途——プログラムを実行する。

書　式——#include <unistd.h>

```
int spawnv (int mode, const char *path, char *const argv[]);
```

解　説——spawnv 関数は現在実行中のプログラムに代わり，*path* で指定したプログラムを *mode* の実行モードで起動し，制御を移す。*mode* に指定できる実行モードは次のとおりである。

- P_OVERLAY　exec 関数の実行と同じ。詳細は execv 関数を参照のこと
- P_WAIT　子プロセスが終了するまで待機する

spawnv 関数には *argv* で，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv* という引数配列が渡されるが，spawnv 関数で指定する *argv* は，この main 関数の *argv* 配列を直接指定しているのと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は *path* と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

戻り値——実行モードが P_OVERLAY ならば，正常に実行できた場合 spawnv 関数は戻ってこない。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。また実行モードが P_WAIT ならば，正常に実行できた場合 spawnv 関数は子プロセスの終了コードを返す。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*path* で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

規　格——***Project LIBC Group***，　***MS–C 7.0***

関連項目——spawnle，spawnlp，spawnve，spawnvp

S

spawnve

用　　途——プログラムを実行する。

書　　式——#include <unistd.h>

```
int spawnve (int mode, const char *path, char *const argv[]
             char *const envp[]);
```

解　　説——spawnve 関数は現在実行中のプログラムに代わり，*path*で指定したプログラムを*mode*の実行モードで起動し，制御を移す。*mode*に指定できる実行モードは次のとおりである。

- P_OVERLAY　　exec 関数の実行と同じ。詳細は execve 関数を参照のこと
- P_WAIT　　子プロセスが終了するまで待機する

spawnve 関数には*argv*で，任意の数の引数列 (文字列の集合) を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動時されると main 関数に *argc*と *argv*という引数配列が渡されるが，spawnve 関数で指定する *argv*は，この main 関数の *argv*配列を直接指定しているのと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は*path*と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

spawnve 関数は実行するプログラムに与える環境エリアを，*envp*の配列で指定することができる (*envp*は引数列の最後に設定される NULL の次である)。実行されるプログラムが，もしも getenv 関数などによって環境変数を検索しようとすると，この配列の内容が検索されることになる。この配列の構造については変数 environ を参照のこと。

戻 り 値——実行モードが P_OVERLAY ならば，正常に実行できた場合 spawnve 関数は戻ってこない。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。また実行モードが P_WAIT ならば，正常に実行できた場合 spawnve 関数は子プロセスの終了コードを返す。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　　引数列が多すぎる
- ENOENT　　*path*で指定されたプログラムが見つからない
- ELOOP　　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　　メモリが足りなくなった

注　　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

規　　格——*Project LIBC Group*，　*MS–C 7.0*

関連項目——spawnl，spawnle，spawnlp，spawnv，spawnvp

spawnvp

用　途——プログラムを実行する。

書　式——#include <unistd.h>

```
int spawnvp (int mode, const char *file, char *const argv[]);
```

解　説——spawnvp 関数は現在実行中のプログラムに代わり，*file*で指定したプログラムを*mode*の実行モードで起動し，制御を移す。*mode*に指定できる実行モードは次のとおりである。

- P_OVERLAY　exec 関数の実行と同じ。詳細は execvp 関数を参照のこと
- P_WAIT　子プロセスが終了するまで待機する

spawnvp 関数には *argv*で，任意の数の引数列（文字列の集合）を渡すことができ，それらは起動されるプログラムに引数として渡される。C のプログラムは起動されると main 関数に *argc* と *argv*という引数配列が渡されるが，spawnvp 関数で指定する *argv*は，この main 関数の *argv*配列を直接指定しているのと同じである。ただし *argv*[0] はプログラム自身を表すので，普通は *file*と同じ値を設定する。また引数配列の最後に，終端記号として NULL を指定すること。

呼び出されるプログラムは *file*で指定する。もしも *file*が "/" や "\" のようなパスの区切り記号を含んでいれば，*file*はプログラムのパスを示しているとみなされる。しかし区切り記号が含まれていなければ，spawnvp 関数は *file*という名前のファイルを環境変数 path に設定されているディレクトリ中から検索し，最初に見つかったものを実行する。

戻り値——実行モードが P_OVERLAY ならば，正常に実行できた場合 spawnvp 関数は戻ってこない。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。また実行モードが P_WAIT ならば，正常に実行できた場合 spawnvp 関数は子プロセスの終了コードを返す。失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　*file*で指定されたプログラムが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——実行しようとするプログラムが **HUPAIR** 仕様に準拠したプログラムならば，引数は **HUPAIR** 仕様にしたがってエンコードされ，プログラムに渡される。逆に，**HUPAIR** 仕様に準拠していないプログラムの場合，**Human68k** で規定されている通常の方法によって引数が渡される。

規　格——*Project LIBC Group*，　*POSIX.1*，　*XPG3*，　*AES/OS*，　*4.3BSD*，　*SYSV*

関連項目——`spawnl`，`spawnle`，`spawnlp`，`spawnv`，`spawnve`

sprintf

用　　途——文字列に対してフォーマット出力を行う。

書　　式——#include <stdio.h>

```
int sprintf (char *s, const char *format, ...);
```

解　　説——sprintf 関数は *format* で指定したフォーマット文字列にしたがって引数を文字列に変換し，その結果を *s* が指す領域に格納する。フォーマット文字列の指定方法や詳細な説明については，fprintf 関数を参照のこと。

戻 り 値——正常に出力できた場合は出力した文字数を返し，失敗した場合は EOF を返す。

注　　意——*s* は結果を格納するだけの十分な大きさの領域を指していなければならない。また，一度に出力できる文字列のサイズは 32K バイトに限定される。その他の制限事項は，fprintf 関数を参照のこと。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——eprintf，fprintf，printf

sqrt

用　途——平方根$\sqrt{x}$を求める。

書　式——
```
#include <math.h>
double sqrt (double x);
#include <sys/xmath.h>
double _f_sqrt (double x);
double _fe_sqrt (double x);
double _fpu_sqrt (double x);
```

解　説——sqrt 関数は x の負でない平方根 (ルート) を求める。各関数はそれぞれ次のように動作する。

- sqrt　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_sqrt　数値演算コプロセッサ命令を直接ドライブする
- _fe_sqrt　FLOAT パッケージを呼び出す
- _fpu_sqrt　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合，sqrt 関数は x の平方根を返す。x が非数 (NaN) または負の値だった場合は，変数 errno に EDOM を設定して同じく非数を返す。

- EDOM　x の値が非数，または x の値が負の値

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは sqrt は _fe_sqrt の別名となり，__DIRECT_IOFPU__が定義されているときは sqrt は _fpu_sqrt の別名となる。また，__DIRECT_FPU__が定義されているときは sqrt は _f_sqrt の別名となる。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——isnan

S

srand

用　　途——乱数シードを初期化する。

書　　式——#include <stdlib.h>

```
void srand (unsigned int seed);
```

解　　説——srand関数は，rand関数やrandom関数で用いられる乱数発生ルーチンで使用される乱数シードを*seed*に設定し，乱数系列を初期化する。rand関数などで得られる乱数列の各要素自体は乱数であり，お互いに相関関係をもたない(実用的範囲で)が，乱数列全体としては乱数シードに対して一定である。

戻り値——なし。

注　　意——srand関数とsrandom関数とは，関数インタフェイスが異なる以外は実質的に同じものであり，たとえばsrand関数はrand関数とともにrandom関数にも影響をおよぼす。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——rand，random，srandom

srandom

用　　途——乱数シードを初期化する。

書　　式——#include <stdlib.h>

```
unsigned int srandom (unsigned int seed);
```

解　　説——srandom 関数は，rand 関数や random 関数で用いられる乱数発生ルーチンで使用される乱数シードを *seed* に設定し，乱数系列を初期化する。rand 関数などで得られる乱数列の各要素自体は乱数であり，お互いに相関関係をもたない (実用的範囲で) が，乱数列全体としては乱数シードに対して一定である。

戻 り 値——変更する前の乱数シードを返す。

注　　意——srand 関数と srandom 関数とは，関数インタフェイスが異なる以外は実質的に同じものであり，たとえば srandom 関数は random 関数とともに rand 関数にも影響をおよぼす。

規　　格——*Project LIBC Group*，　*4.3BSD*

関連項目——rand，random，srand

S

sscanf

用　　途――文字列からフォーマット入力を実行する。

書　　式――#include <stdio.h>

```
int sscanf (const char *s, const char *format, ...);
```

解　　説――sscanf 関数は，*format* で指定した入力フォーマット文字列にしたがって必要なデータ列を *s* で指定した文字列から入力し，その結果を引数の指す領域に格納する。入力フォーマット文字列の指定方法や詳細な説明については，fscanf 関数を参照のこと。

戻 り 値――正常に入力できた場合は実際に入力できた項目数を返し，失敗した場合は EOF を返す。

注　　意――引数に指定するポインタは，結果を格納するための十分な大きさの領域を指していなければならない。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――fscanf，scanf

stat

用　途——ファイルのステータス情報を取得する。

書　式——

```
#include <stat.h>
int stat (const char *name, struct stat *st);
int fstat (int fno, struct stat *st);
int lstat (const char *name, struct stat *st);
```

解　説——stat 関数と lstat 関数は，*name* の指すファイルのステータス情報を取得し，その結果を *st* ポインタで指される構造体に代入する。また fstat 関数は，*fno* で指定したファイルハンドルが指すファイルのステータスを取得し，その結果を *st* ポインタで指される構造体に代入する。

シンボリックリンクはファイルのステータスを取得する前に解決されるが，lstat 関数を用いれば，シンボリックリンクを解決せずにリンクファイル自体の情報を取得することができる。

stat 構造体の構造は次のとおり。まず，対象が通常のファイルまたはディレクトリである場合。

```
struct stat {
    dev_t    st_dev;     /* 物理ドライブ番号 */
    ino_t    st_ino;     /* 物理ドライブ× 16777216 ＋先頭セクタ番号 */
    mode_t   st_mode;    /* ファイルモード */
    nlink_t  st_nlink;   /* つねに 1 */
    uid_t    st_uid;     /* つねに 0(root) */
    gid_t    st_gid;     /* つねに 0(root) */
    dev_t    st_rdev;    /* 論理ドライブ番号 */
    off_t    st_size;    /* ファイルサイズ，ディレクトリサイズ */
    time_t   st_atime;   /* 最終アクセス時間 */
    time_t   st_mtime;   /* 最終変更時間 */
    time_t   st_ctime;   /* 最終作成時間 */
};
```

- st_ino は物理ドライブ番号を 24 ビットシフトし，対象となるファイルの先頭セクタ番号を加えた値を返す
- st_mode は拡張 UNIX ファイルモードの値を返す
- st_size はターゲットがファイルの場合はファイルサイズを返し，ディレクトリの場合はディレクトリエントリのサイズを返す

S

次に対象がキャラクタ型デバイスである場合。

```
struct stat {
    dev_t   st_dev;     /* デバイスドライバの常駐アドレス */
    ino_t   st_ino;     /* つねに 0 */
    mode_t  st_mode;    /* つねに S_IFCHR | S_IWRITE | S_IREAD */
    nlink_t st_nlink;   /* つねに 1 */
    uid_t   st_uid;     /* つねに 0(root) */
    gid_t   st_gid;     /* つねに 0(root) */
    dev_t   st_rdev;    /* デバイスドライバの常駐アドレス */
    off_t   st_size;    /* つねに 0 */
    time_t  st_atime;   /* つねに 0(1970 年 1 月 1 日) */
    time_t  st_mtime;   /* つねに 0(1970 年 1 月 1 日) */
    time_t  st_ctime;   /* つねに 0(1970 年 1 月 1 日) */
};
```

- 標準入出力および標準エラー出力，AUX，PRN にリンクされているキャラクタ型デバイスドライバにかぎり，情報を得ることができる。また，シンボリックリンクのリンク先がキャラクタ型デバイスの場合も可能
- st_dev および st_rdev はデバイスドライバが常駐している先頭アドレスを返す
- st_mode は拡張 **UNIX** ファイルモードの値を返す

拡張 **UNIX** ファイルモードの値は次のように定義されている。まず，ファイルモードの下位 16 ビットは，一般に **UNIX** で定義されているファイルモードとビットレベルで完全に互換であるようになっている。

S_IXOTH	00000001	第 3 者実行
S_IWOTH	00000002	第 3 者書き込み
S_IROTH	00000004	第 3 者読み込み
S_IXGRP	00000010	グループ実行
S_IWGRP	00000020	グループ書き込み
S_IRGRP	00000040	グループ読み込み
S_IXUSR	00000100	オーナー実行
S_IWUSR	00000200	オーナー書き込み
S_IRUSR	00000400	オーナー読み込み
S_ISVTX	00001000	スティッキービット (つねに 0)
S_ISGID	00002000	Set GID ビット (つねに 0)
S_ISUID	00004000	Set UID ビット (つねに 0)

下位 16 ビット内でも，次の値はビット値の論理和で設定されているわけではないので，注意が必要である。これらの値をチェックするには，ファイルモードに対して S_IFMT でマスクをかけてから等号で行う。決して，論理積でチェックしてはいけない。またこれらの目的のために，S_ISCHR や S_ISDIR などのマクロがある。詳細は<sys/stat.h>を参照のこと。

S_IFIFO	00010000	FIFO (つねに 0)
S_IFCHR	00020000	キャラクタデバイス
S_IFDIR	00040000	ディレクトリ
S_IFBLK	00060000	ブロックデバイス
S_IFREG	00100000	通常ファイル
S_IFLNK	00120000	シンボリックリンク
S_IFSOCK	00140000	ソケット (つねに 0)

次に上位 16 ビットだが，こちらは **Human68k** の特殊な属性情報を管理するために拡張されている。通常 **UNIX** のファイルモードは 16 ビットなので，この部分がアクセスされることはない。

S_IRDONLY	00200000	リードオンリー
S_IFVOL	00400000	ボリュームファイル
S_ISYS	01000000	システムファイル
S_IHIDDEN	02000000	隠しファイル
S_IEXBIT	04000000	実行ビット

- 実行属性/書き込み属性はオーナー/グループ/第 3 者すべて同じ値が設定される
- 読み込み属性はつねに有効になる

戻 り 値——正常に情報を取得できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　ファイルが見つからない

注　　意——st_dev の値は，仮想ドライブ，仮想ディレクトリ，シンボリックリンクを展開した値が設定されるが，シンボリックリンクファイルのネストは 1 段までしか参照しないため，完全には物理ドライブにならない場合もありえる。また，st_ino も同じである。

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——_mode2dos，_mode2unix，chmod，fchmod，open，symlink，utime

S

strcat

用　　途——文字列をほかの文字列に連結する。

書　　式——#include <string.h>

```
char *strcat(char *string1, const char *string2);
```

解　　説——strcat 関数は *string2* の指す文字列を，*string1* の指す文字列の最後にコピーし，連結した文字列の最後に null 文字を追加する。*string1* の末尾の null 文字には，*string2* の最初の文字を重ねてコピーする。

戻 り 値——*string1* へのポインタを返す。

注　　意——*string1* は，連結した結果の文字列を格納するのに十分な領域を指していなければならない。1 文字は 1 バイトを意味する。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——strncat

strchr

用　途——文字列中から指定文字を検索する。

書　式——#include <string.h>

```
char *strchr (const char *string, int character);
```

解　説——strchr 関数は *string* が指す文字列中から，*character* で指定された文字を検索する。*string* 末尾の null 文字も検索の対象となり得る。*character* は，内部で int 型から char 型に変換される。

戻り値——*string* 中で最初に現れた *character* へのポインタを返し，見つからない場合は NULL を返す。

注　意——1 文字は 1 バイトを意味する。

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——memchr，strcspn，strpbrk，strrchr，strspn，strstr，strtok

strcmp

用　　途——2つの文字列を比較する。

書　　式——#include <string.h>

```
int strcmp (const char *string1, const char *string2);
```

解　　説——strcmp 関数は *string1* の指す文字列と *string2* の指す文字列を比較する。比較は null 文字が検出された時点で終了する。

戻 り 値——比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意——1文字は1バイトを意味する。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——memcmp，strcoll，strncmp，strxfrm

strcmpi

用　　途――2つの文字列を大文字と小文字を区別しないで比較する。

書　　式――#include <string.h>

```
int strcmpi (const char *string1, const char *string2);
```

解　　説――strcmpi 関数は *string1* の指す文字列と *string2* の指す文字列を，大文字/小文字を区別しないで比較する。比較は null 文字が検出された時点で終了する。

戻 り 値――比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意――strcmpi 関数は文字列を小文字にしてから比較する。strcmpi 関数はマクロ定義であり，実体は stricmp 関数である。1文字は1バイトを意味する。

規　　格――***Project LIBC Group***，***4.3BSD***

関連項目――memcmp，strcmp，strcoll，stricmp，strncmp，strxfrm

S

strcoll

用　　途――2つの文字列をロケールを用いて比較する。

書　　式――#include <string.h>

```
int strcoll (const char *string1, const char *string2);
```

解　　説――strcoll 関数は *string1* の指す文字列と *string2* の指す文字列を，現在のロケールの LC_COLLATE カテゴリを基に比較する。比較は null 文字が検出された時点で終了する。

戻 り 値――比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意――strcoll 関数はCロケールにしか対応していないため，strcmp 関数と同じである。1文字は1バイトを意味する。

規　　格――*Project LIBC Group*，*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――strlwr，strupr，strxfrm

strcpy

用　途——文字列をコピーする。

書　式——#include <string.h>

```
char *strcpy (char *string1, const char *string2);
```

解　説——strcpy 関数は *string2* の指す文字列を，null 文字を含めて *string1* の指す領域にコピーする。

戻り値——*string1* へのポインタを返す。

注　意——領域が重なっていた場合の動作は未定義である。また *string1* は，*string2* の指す文字列を格納するのに十分な領域を指していなければならない。1 文字は 1 バイトを意味する。

規　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***SYSV***

関連項目——memcpy，memset，strncpy

S

strcspn

用　　途――指定文字列に含まれない文字が，ほかの文字列の先頭から何文字続いているかを調べる。

書　　式――#include <string.h>

```
size_t strcspn (const char *string1, const char *string2);
```

解　　説――strcspn関数は*string1*の指す文字列と*string2*の指す文字列を比較し，*string1*の先頭から*string2*に含まれない文字で構成されている部分の長さを調べる。

戻 り 値――*string1*の先頭から*string2*に含まれない文字で構成されている部分の長さを返す。

注　　意――1文字は1バイトを意味する。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目――memchr，strchr，strpbrk，strrchr，strspn，strstr，strtok

strdup

用　　途——新しい領域を確保して文字列をコピーする。

書　　式——#include <string.h>

```
char *strdup (const char *string);
```

解　　説——strdup 関数は *string* の指す文字列を，null 文字を含めて malloc 関数によって確保した領域にコピーする。

戻 り 値——正常にコピーできた場合はコピー先へのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　　意——1 文字は 1 バイトを意味する。

規　　格——*SYSV*

関連項目——malloc，strcpy

strerror

用　　途——エラーメッセージへのポインタを返す。

書　　式——
```
#include <string.h>
char *strerror (int error);
```

解　　説——strerror 関数は，*error* で示されるエラーコードに対応するエラーメッセージへのポインタを返す。

戻 り 値——エラーメッセージへのポインタを返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定して NULL を返す。

- EINVAL　*error* の値が無効である

規　　格——*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目——memcpy，memset，strncpy

strftime

用　途──詳細時間の情報を文字列に変換する。

書　式──
```
#include <time.h>
size_t strftime (char *s, size_t maxsize,
                 const char *format, const struct tm *timeptr);
```

解　説──strftime 関数は *timeptr* で与えられる**詳細時間**の情報を，*format* で指定したフォーマットで文字列に変換し，その結果を *s* の指す領域に格納する。*s* の指す領域のサイズは *maxsize* で制限され，変換結果がこのサイズをこえるようならば，*maxsize* バイトに収まる範囲までしか変換が行われない。

format に指定するフォーマット文字は通常の文字列であるが，“%” で表される**変換指定文字**に続くアルファベット 1 文字があると，その指示にしたがって変換が行われる。これらの変換は，現在の**ロケール**の LC_TIME カテゴリに左右される。

変換指示文字には次のものが使用できる。

- %a　ロケールで定義される曜日名の省略形
- %A　ロケールで定義される曜日名
- %b　ロケールで定義される月の名前の省略形
- %B　ロケールで定義される月の名前
- %c　ロケールで定義される日付/時間の表現
- %d　日の数字表現 ($01 \le x \le 31$)
- %D　日付 (%m/%d/%y)
- %h　ロケールで定義される月の名前の省略形
- %H　時の数字表現 (24 時間制 $00 \le x \le 23$)
- %I　時の数字表現 (12 時間制 $00 \le x \le 12$)
- %j　通算日の数字表現 ($001 \le x \le 366$)
- %m　月の数字表現 ($01 \le x \le 12$)
- %M　分の数字表現 ($00 \le x \le 59$)
- %n　改行文字
- %p　ロケールで定義される “AM”，“PM” 相当の文字列
- %r　12 時間制の時間表現 (英米国式) (%I:%M:%S [AM|PM])
- %S　秒の数字表現 ($00 \le x \le 61$)
- %t　タブ文字
- %T　時間表現 (%H:%M:%S)
- %U　日曜を週の始めとした通算週 ($00 \le x \le 53$)
- %w　曜日の数字表現 (日曜が 0 $0 \le x \le 6$)
- %W　月曜を週の始めとした通算週 ($00 \le x \le 53$)

S

- %x　ロケールで定義される日付表現
- %X　ロケールで定義される時間表現
- %y　世紀を省いた年の数字表現 ($00 \leq x \leq 99$)
- %Y　世紀を伴った年の数字表現 (ex. 1993)
- %Z　地域時間の時間帯名称
- %%　%文字

戻り値——正常に変換できた場合は*s*に格納したバイト数 (終端のnull文字は含まない) を返し，失敗した場合には0を返す。

注　意——*s*は結果を格納するための十分な大きさの領域を指していることが望ましい。

互換性——*LIBC*はCロケールしかサポートしていない。

規　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV***

関連項目——`ctime`，`tzset`

stricmp

用　途——2つの文字列を大文字と小文字を区別しないで比較する。

書　式——`#include <string.h>`

```
int stricmp (const char *string1, const char *string2);
```

解　説——stricmp 関数は *string1* の指す文字列と *string2* の指す文字列を大文字と小文字を区別しないで比較する。比較は null 文字が検出された時点で終了する。

戻り値——比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　意——stricmp 関数は文字列を小文字にしてから比較する。1文字は1バイトを意味する。

規　格——***Project LIBC Group***，　***4.3BSD***

関連項目——memcmp，strcmp，strcmpi，strcoll，strncmp，strxfrm

S

strlen

用　　途——文字列の長さを調べる。

書　　式——#include <string.h>

```
size_t strlen (const char *string);
```

解　　説——strlen関数は，*string*の指す文字列の末尾のnull文字を除いた文字列の長さを調べる。

戻 り 値——末尾のnull文字を除いた文字列の長さを返す。

注　　意——1文字は1バイトを意味する。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——strcpy

strlwr

用　　途——文字列を小文字に変換する。

書　　式——`#include <string.h>`

```
char *strlwr (char *string);
```

解　　説——`strlwr` 関数は *string* が指す文字列を先頭から調べ，現在の**ロケール**の LC_CTYPE カテゴリをもとに大文字を小文字に変換する。処理は null 文字を検出した時点で終了する。

戻 り 値——*string* へのポインタを返す。

注　　意——*LIBC* は C ロケールしかサポートしていない。1 文字は 1 バイトを意味する。

規　　格——***Project LIBC Group***，　***MS–C 7.0***

関連項目——`strnset`，`strrev`，`strset`，`strupr`

S

strncat

用　　途――文字列を指定文字数だけほかの文字列につけ加える。

書　　式――#include <string.h>

```
char *strncat (char *string1, const char *string2, size_t n);
```

解　　説――strncat 関数は，*string2* の指す文字列を *string1* の指す文字列の最後にコピーし，連結した文字列の最後に null 文字を追加する。*string1* の末尾の null 文字には，*string2* の最初の文字を重ねてコピーする。コピーは null 文字が検出されるか，*n* 文字コピーした時点で終了する。

戻 り 値――*string1* へのポインタを返す。

注　　意――*string1* は，連結した結果の文字列を格納するのに十分な領域を指していなければならない。1文字は1バイトを意味する。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――strcat

strncmp

用　途——2つの文字列を指定文字数だけ比較する。

書　式——#include <string.h>

```
int strncmp (const char *string1, const char *string2, size_t n);
```

解　説——strncmp 関数は，*string1* の指す文字列と *string2* の指す文字列を比較する。比較は null 文字が検出されるか，*n* 文字比較した時点で終了する。

戻り値——比較の結果，2つの文字列がまったく同じならば0を返す。異なっていたり *n* 文字を比較し終えた場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　意——1文字は1バイトを意味する。

規　格——***ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV***

関連項目——memcmp，strcmp，strcoll，strxfrm

S

strncpy

用　　途——文字列を指定文字数だけコピーする。

書　　式——#include <string.h>

```
char *strncpy (char *string1, const char *string2, size_t n);
```

解　　説——strncpy 関数は *string2* の指す文字列の最初の *n* 文字までを，*string1* の指す領域にコピーする。コピーは null 文字が検出されるか，*n* 文字コピーした時点で終了する。

戻 り 値——*string1* へのポインタを返す。

注　　意——領域が重なっていた場合の動作は未定義である。また *string1* は，*n* 文字を格納するのに十分な領域を指していなければならない。1文字は1バイトを意味する。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——memcpy，memset，strcpy

strnset

用　途——文字列を指定文字で指定文字数分だけ埋める。

書　式——

```
#include <string.h>
char *strnset (char *string, int character, size_t n);
```

解　説——strnset 関数は，*string* が指す文字列の先頭から *n* 文字か null 文字を検出するまで，*character* で指定された文字で埋める。*character* は，内部で int 型から char 型に変換される。

戻り値——*string* へのポインタを返す。

注　意——*n* は文字列領域の大きさを越えてはならない。1 文字は 1 バイトを意味する。

規　格——*MS–C 7.0*

関連項目——memset，strlen，strlwr，strrev，strset，strupr

S

strpbrk

用　　途――指定文字列に含まれる文字がほかの文字列に存在するかどうかを調べる。

書　　式――#include <string.h>

```
char *strpbrk (const char *string1, const char *string2);
```

解　　説――strpbrk 関数は *string1* の指す文字列と *string2* の指す文字列を比較し，*string2* に含まれる文字が最初に現れる場所を検索する。

戻 り 値――*string1* 中で *string2* に含まれる文字が最初に現れる位置へのポインタを返し，見つからない場合は NULL を返す。

注　　意――1文字は1バイトを意味する。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目――memchr，strchr，strcspn，strrchr，strspn，strstr，strtok

strrchr

用　　途——文字列中から指定文字が最後に現れる位置を検索する。

書　　式——#include <string.h>

```
char *strrchr (const char *string, int character);
```

解　　説——strrchr 関数は *string* が指す文字列中から，*character* で指定された文字が最後に現れる位置を検索する。*string* 末尾の null 文字も検索の対象となり得る。*character* は，内部で int 型から char 型に変換される。

戻 り 値——*string* 中で最後に現れる *character* へのポインタを返す。見つからない場合は NULL を返す。

注　　意——1 文字は 1 バイトを意味する。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——memchr，strchr，strcspn，strpbrk，strspn，strstr，strtok

strrev

用　　途——文字列を前後反転させる。

書　　式——

```
#include <string.h>
char *strrev (char *string);
```

解　　説——strrev 関数は *string* が指す文字列を前後反転させる。

戻 り 値——*string* へのポインタを返す。

注　　意——1文字は1バイトを意味する。

規　　格——*MS-C 7.0*

関連項目——strlwr, strnset, strset, strupr

strset

用　　途——文字列を指定文字で埋める。

書　　式——#include <string.h>

```
char *strset (char *string, int character);
```

解　　説——strset 関数は *string* が指す文字列の先頭から，null 文字に出会うまで *character* で指定された文字で埋める。*character* は，内部で int 型から char 型に変換される。

戻 り 値——*string* へのポインタを返す。

注　　意——1 文字は 1 バイトを意味する。

規　　格——*MS–C 7.0*

関連項目——strlwr，strnset，strrev，strupr

strsignal

用　　途――シグナルを表す文字列を取得する。

書　　式――#include <signal.h>

```
char *strsignal (int signo);
```

解　　説――strsignal 関数は，*signo* で指定されたシグナルを表す文字列へのポインタを返す。たとえば，*signo* に SIGABRT を指定すると，以下のような文字列へのポインタが返る。

“SIGABRT Abnormal termination”

戻 り 値――正常に取得できた場合は文字列へのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正なシグナル *signo* を指定した

規　　格――*Project LIBC Group*，*4.3BSD*

関連項目――signal，strerror

strspn

用　途——指定文字列に含まれる文字が，ほかの文字列の先頭から何文字続いているかを調べる。

書　式——

```
#include <string.h>
size_t strspn (const char *string1, const char *string2);
```

解　説——strspn 関数は *string1* の指す文字列と *string2* の指す文字列を比較し，*string1* の先頭から *string2* に含まれる文字で構成されている部分の長さを調べる。

戻り値——*string1* の先頭から *string2* に含まれる文字で構成されている部分の長さを返す。

注　意——1 文字は 1 バイトを意味する。

規　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——memchr，strchr，strcspn，strpbrk，strrchr，strstr，strtok

S

strstr

用　　途——指定文字列がほかの文字列に存在するかどうかを調べる。

書　　式——#include <string.h>

```
size_t strstr (const char *string1, const char *string2);
```

解　　説——strstr 関数は *string1* の指す文字列と *string2* の指す文字列を比較し，*string1* 中で最初に文字列 *string2* が現れる位置を検索する。

戻 り 値——*string1* 中で，最初に *string2* が現れる位置へのポインタを返す。

注　　意——1文字は1バイトを意味する。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——memchr，strchr，strcspn，strpbrk，strrchr，strspn，strtok

strtod

用　途——文字列を double 型倍精度浮動小数に変換する。

書　式——#include <stdlib.h>

```
double strtod (const char *nptr, char **endptr);
```

解　説——strtod 関数は *nptr* で指定された文字列を，まず 3 つの部分文字列，すなわち先頭の空白部分，浮動小数部分，認識不能部分 (終端の null 文字を含む) に分割する。strtod 関数はこれらの部分文字列のうち，浮動小数部分の文字列を double 型倍精度浮動小数に変換し，その値を返す。この浮動小数部分は符号 (省略可能)，浮動小数および指数 (省略可能) からなり，指数部分は “e” か “E” に続いて符号 (省略可能) と指数値からなる。

```
-3.1415926, 1.8794E+15, 2.42e-6
```

もし *endptr* が NULL でない場合は，*endptr* に最後の認識不能部分へのポインタを格納する。この部分は終端の null 文字を含むので，1 文字以上である。小数点として認識する文字はロケールの LC_NUMERIC カテゴリに影響される。

戻り値——変換した結果を返す。ただし結果がオーバーフローした場合は HUGE_VAL を返し，アンダーフローした場合は 0 を返して，変数 errno にその原因を示すエラーコードを設定する。

nptr が空文字であったり，すべて空白だったり，認識不能文字だったりして浮動小数部分の文字列が見つからない場合変換は行われず，0 を返す。また，このとき *endptr* には *nptr* の値が格納される。

- ERANGE　オーバーフロー，あるいはアンダーフローを起こした

S

注　意——*LIBC* は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目——atof，atoi，atol，strtol，strtoul

strtok

用　　途――文字列を指定した区切文字でトークンに分ける。

書　　式――#include <string.h>

```
char *strtok (char *string, const char *delim);
```

解　　説――strtok関数を連続して呼ぶことにより，*string*が指す文字列を語句(トークン)に分解することができる。各語句は，*delim*が指す文字列に含まれる区切り文字(デリミタ)によって区切られる。

最初にstrtok関数を呼び出すときは，デリミタ文字列を*string*に指定し，以降strtok関数を呼び出すときには，*string*にNULLを指定する。デリミタ文字は，strtok関数を呼び出すたびに変えることが可能である。*string*にデリミタ文字が続く場合は無視される。

戻り値――語句が見つかれば最後の区切り記号の位置にnull文字を設定し，語句の最初の文字へのポインタを返す。見つからなければNULLを返す。

注　　意――strtok関数は与えられた*string*の領域を書き換えるので，元のデータが必要な場合は自分で保存しておく必要がある。1文字は1バイトを意味する。

規　　格――*ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *SYSV*

関連項目――memchr, strchr, strcspn, strpbrk, strrchr, strspn, strstr

strtol

用　途——文字列を long 型整数に変換する。

書　式——#include <stdlib.h>

```
long strtol (const char *nptr, char **endptr, int radix);
```

解　説——strtol 関数は *nptr* で指定された文字列を，まず3つの部分文字列，すなわち先頭の空白部分，整定数部分，認識不能部分 (終端の null 文字を含む) に分割する。strtol 関数はこれらの部分文字列のうち，整定数部分の文字列を，*radix* で指定した値を基数として long 型整数に変換し，その値を返す。

整定数部分は符号 (省略可能) から始まり，基数が 2 ～ 10 ならば 0 ～ 9 の文字を，また基数が 10 ～ 36 ならばこれに加えて A ～ Z および a ～ z の文字が数値として認識される。

もし基数 *radix* が 0 の場合，入力文字列の基数は 8，10，16 のいずれかであり，文字列の特徴から動的に決定される。すなわち，01 ～ 07 から始まれば 8，0x から始まれば 16，それ以外なら 10 である。また基数が 16 の場合は，接頭辞 0x も数値の一部として認識される。

ただし数値の最後の u，U，l，L などの unsigned 型指定文字や long 型指定文字は，数値としては認識されないので注意すること。

endptr が NULL でない場合は，*endptr* に最後の認識不能部分へのポインタを格納する。この部分は終端の null 文字を含むので，1 文字以上である。

戻り値——変換した結果を返す。ただし変換結果が long 型で表現しきれない場合は，符号に応じて LONG_MAX あるいは LONG_MIN を返し，変数 errno にその原因を示すエラーコードを設定する。

nptr が空文字であったり，すべて空白だったり，認識不能文字だったりして整定数部分の文字列が見つからない場合変換は行われず，0 を返す。また，このとき *endptr* には *nptr* の値が格納される。

- ERANGE　変換結果が long 型で表現することができない

注　意——*LIBC* は C ロケールしかサポートしていない。

規　格——***Project LIBC Group***，***ANSI C***，***XPG3***，***AES/OS***，***SYSV***

関連項目——atof，atoi，atol，strtod，strtoul

S

strtoul

用　　途――文字列を unsigned long 型整数に変換する。

書　　式――#include <stdlib.h>

```
unsigned long strtoul (const char *nptr, char **endptr,
                       int radix);
```

解　　説――strtoul 関数は *nptr* で指定された文字列を，まず3つの部分文字列，すなわち先頭の空白部分，整定数部分，認識不能部分 (終端の null 文字を含む) に分割する。strtoul 関数はこれらの部分文字列のうち，整定数部分の文字列を，*radix* で指定した値を基数として unsigned long 型整数に変換し，その値を返す。

整定数部分は符号 (省略可能) から始まり，基数が 2 ～ 10 ならば 0 ～ 9 の文字を，また基数が 10 ～ 36 ならばこれに加えて A ～ Z および a ～ z の文字が数値として認識される。

もし基数 *radix* が 0 の場合は，入力文字列の基数は 8，10，16 のいずれかであり，文字列の特徴から動的に決定される。すなわち，01 ～ 07 から始まれば 8，0x から始まれば 16，それ以外なら 10 である。また基数が 16 の場合は，接頭辞 0x も数値の一部として認識される。

ただし数値の最後の u，U，l，L などの unsigned 型指定文字や long 型指定文字および負の符号 "-" は，数値としては認識されないので注意すること。

endptr が NULL でない場合は，*endptr* に最後の認識不能部分へのポインタを格納する。この部分は終端の null 文字を含むので，1 文字以上である。

戻 り 値――変換した結果を返す。ただし変換結果が unsigned long 型で表現しきれない場合は ULONG_MAX を返し，変数 errno にその原因を示すエラーコードを設定する。

nptr が空文字であったり，すべて空白だったり，認識不能文字だったりして整定数部分の文字列が見つからない場合変換は行われず，0 を返す。また，このとき *endptr* には *nptr* の値が格納される。

- ERANGE　変換結果が unsigned long 型で表現することができない

注　　意――*LIBC* は C ロケールしかサポートしていない。

規　　格――***Project LIBC Group***，***ANSI C***，***XPG3***，***AES/OS***，***SYSV***

関連項目――atof，atoi，atol，strtod，strtol

strupr

用　途——文字列を大文字に変換する。

書　式——
```
#include <string.h>
char *strupr (char *string);
```

解　説——strupr 関数は *string* が指す文字列を先頭から調べ，現在のロケールの LC_CTYPE カテゴリをもとに小文字を大文字に変換する。処理は null 文字を検出した時点で終了する。

戻り値——*string* へのポインタを返す。

注　意——*LIBC* は C ロケールしかサポートしていない。1 文字は 1 バイトを意味する。

規　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——strlwr，strnset，strrev，strset

S

strxfrm

用　　途――2つの文字列をロケールを用いて指定文字数だけコピーする。

書　　式――#include <string.h>

```
size_t strxfrm (char *string1, const char *string2, size_t n);
```

解　　説――strxfrm 関数は *string2* が指す文字列の最初から *n* 文字を，現在のロケールの LC_COLLATE カテゴリをもとに変換し，*string1* の指す領域にコピーする。コピーは null 文字が検出されるか，*n* 文字コピーした時点で終了する。*string2* 末尾の null 文字も変換の対象となりうる。また *n* が 0 の場合は，*string1* は NULL でもよい。

戻 り 値――変換された文字のバイト数 (null 文字は含まない) を返すが，*n* が 0 の場合は単に *string2* の長さを返す。

注　　意――領域が重なっていた場合の動作は未定義である。また *string1* は，*string2* を変換した文字列を格納するのに十分な領域を指していなければならない。strxfrm 関数は C ロケールにしか対応していないため，基本的には strncpy 関数と同じである。1 文字は 1 バイトを意味する。

規　　格――***Project LIBC Group***，　***ANSI C***，　***XPG3***，　***AES/OS***，　***SYSV***

関連項目――strcoll，strlwr，strupr

swab

用　途——文字を交換する。

書　式——#include <string.h>

```
void swab (const char *src, char *dst, size_t n)
```

解　説——swab 関数は *src* が指す領域から *n* バイトを *dst* が指す領域にコピーし，隣り合った偶数バイトと奇数バイトを入れ換える。*n* が偶数でない場合は，*n* を越えない最大の偶数に丸められる。

戻り値——なし。

注　意——領域が重なっていた場合の動作は未定義である。swab 関数は Big-Endian と Little-Endian とのマシン間で，バイナリデータを転送するのに用いられる。1文字は1バイトを意味する。

規　格——***POSIX.1***，***4.3BSD***

関連項目——memcpy

S

symlink

用　　途――シンボリックリンクファイルを作成する。

書　　式――

```
#include <dos.h>
int symlink (const char *src, const char *dst);
```

解　　説――symlink 関数は，リンク先として *src* を指すようなシンボリックリンクファイル *dst* を作成する。ただし，*dst* と同名のファイルが存在する場合は失敗する。

戻 り 値――正常にシンボリックリンクファイルができた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EEXIST　　*dst* がすでに存在している
- ELOOP　　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOSPC　　ディスクがいっぱいである
- ENOSYS　　**lndrv** が常駐していない

注　　意――symlink 関数の動作はすべて **lndrv** に依存するため，**lndrv** 以外のシンボリックリンクドライバでは使用することができない。また将来，**lndrv** が変更されたりしても動作しなくなる可能性がある。

規　　格――***Project LIBC Group***，***AES/OS***，***4.3BSD***

関連項目――_dos_importlnenv，_dos_lfiles，_dos_readlink，readlink

sysconf

用　途——システムに関する情報を取り出す。

書　式——
```
#include <unistd.h>
long sysconf (int name);
```

解　説——sysconf 関数は現在のシステム (およびライブラリ) に関連する情報を取得し，ユーザアプリケーションに対してシステム設定値などを得る手段を与える。どの設定値を調べるかについては，*name*で指定する。*name*に指定することができるマクロ値と求める設定値の一覧は次のとおり。

- _SC_ARG_MAX　exec 関数に渡すことができる引数の最大値
- _SC_CHILD_MAX　1 ユーザ当たりの子プロセスの最大数 (参考値)
- _SC_CLK_TCK　1 秒当たりのクロックティック
- _SC_NGROUPS_MAX　1 プロセス当たりの予備グループ ID の最大数
- _SC_OPEN_MAX　オープンできるファイルハンドルの最大数
- _SC_JOB_CONTROL　ジョブコントロールをサポートしているか否か
- _SC_SAVED_IDS　SAVED ID をサポートしているか否か
- _SC_VERSION　*POSIX.1*のバージョン番号
- _SC_PASS_MAX　パスワードの最大長
- _SC_XOPEN_VERSION　*XPG3*のバージョン番号
- _SC_ATEXIT_MAX　atexit 関数で登録できる関数の最大数
- _SC_PAGE_SIZE　1 ページのページサイズ
- _SC_AES_OS_VERSION　*AES/OS*のバージョン番号
- _SC_PROJ_VERSION　ライブラリのバージョン番号

戻り値——正常に情報を取得できた場合はその値を返し，失敗した場合には–1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な *name*を指定した

互換性——*LIBC*では上記 (詳細は<limits.h>を参照) の制限値は可変ではなく，すべて固定である。

規　格——*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目——fpathconf，pathconf

S

system

用　途——シェルコマンドを実行する。

書　式——#include <stdlib.h>

```
#include <stdlib.h>
int system (const char *command);
extern int _keep_cwd_on_exec;
```

解　説——system 関数は，*command* で指定されたコマンドをシェルに与え，そのコマンドを端末から入力したかのように実行させる。*command* に含まれる文字列は，通常，シェルにとって意味のある文字 (パイプやリダイレクトなど) を含むことができる。また，*command* が NULL あるいは空文字を指していれば，シェルを対話的シェル (*Interactive shell*) として起動する。

system 関数を実行したプログラムは，子プロセスであるシェル，およびそのシェルから実行されたプログラムの実行が終了するまで停止する。

起動するシェルは対話的に起動するかどうかによって変更することができる。

- **対話的**　環境変数 SHELL の指すプログラム。シェルに対する先行引数 (後述) は環境変数 SHELL_OPT から取得し，シェルのタイプ (後述) は環境変数 SHELLTYPE から取得する
- **非対話的**　環境変数 SYSTEM_SHELL の指すプログラム。シェルに対する先行引数は環境変数 SYSTEM_SHELL_OPT から取得し，シェルのタイプは環境変数 SYSTEM_SHELLTYPE から取得する
  ただし環境変数 SYSTEM_SHELL が定義されていない場合，シェル/先行引数/シェルタイプは対話的シェルの場合と同様に，それぞれ環境変数 SHELL，SHELL_OPT，SHELLTYPE が使われる。

○ **シェルのタイプ**

シェルの起動方法はシェルのタイプが “COMMAND.X” タイプか，**UNIX** タイプかによって異なる。system 関数は，起動するシェルのタイプを以下のようにして判定する。

1. 環境変数 SHELLTYPE(非対話的シェルの場合は，SYSTEM_SHELLTYPE) の値が “COMMAND” ならば “COMMAND.X” タイプ，“UNIX” あるいはそれ以外ならば **UNIX** タイプとする。
2. 環境変数 SHELLTYPE が未定義の場合，環境変数 SHELL(非対話的シェルの場合は SYSTEM_SHELL) が未定義ならば “COMMAND.X” タイプ，環境変数 SHELL の設定の末尾が “command” あるいは “command.x” ならば “COMMAND.X” タイプ，それ以外は **UNIX** タイプとする (ただし文字列の比較は，大文字/小文字を無視して行われる)。

○ **非対話的シェルの起動**

1. “COMMAND.X” タイプシェルの場合，以下のようなコマンドラインを **Human68k** で規定されている通常の方法で子プロセスに渡す。

   COMMAND /C *command*

2. **UNIX** タイプシェル (たとえば “fish”) の場合，以下のようなコマンドラインを **HUPAIR** で規定された仕様にしたがってエンコードし，子プロセスに渡す (*command* は単一の引数 (*argv*[2]) として渡される)。

   fish -c *command*

この “/C” や “-c” は，前述の「シェルに対する先行引数」を設定する環境変数 SHELL_OPT(非対話的シェルの場合は SYSTEM_SHELL_OPT) によって変更することができる。

戻り値——正常に実行できた場合はシェルの終了コードを返し，失敗した場合には 127 << 8 (32512) を返して，変数 errno にその原因を示すエラーコードを設定する。

- E2BIG　引数列が多すぎる
- ENOENT　シェルが見つからない
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている
- ENOMEM　メモリが足りなくなった

注　意——環境変数 SHELL が未定義ならば，“COMMAND.X” がシェルとして使われる。

通常，環境変数 SHELL にはシェルのファイルネームをフルパスで設定するが，そうでない場合には環境変数 path に設定されたディレクトリが検索され，最初に見つかったものを対象とする。

system 関数はシェルを起動する際にカレントディレクトリを保存する。これを避けたい場合には，変数_keep_cwd_on_exec に 0 を設定すること。また system 関数は，シェルを起動する際に標準ファイルハンドル (0 ～ 4) をすべて保存する。

system 関数を非対話的に使うプログラムは，なるべく起動するシェルに依存しないようなコマンドラインを渡すべきである。

互換性——**UNIX** では，system 関数は環境変数 SHELL の設定に関わらず “/bin/sh” を起動する。もしも同じような環境にしたいのならば，環境変数 SYSTEM_SHELL, SYSTEM_SHELL_OPT，SYSTEM_SHELLTYPE で実現できる。

規　格——***ANSI C***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——spawnve，spawnvp

S

_tobslash

用　　途――パス名に含まれるパス区切り記号をバックスラッシュに変換する。

書　　式――#include <sys/xglob.h>

```
char *_tobslash (char *path);
```

解　　説――_tobslash関数は*path*で指定されたパス名に含まれるすべてのパス区切り記号を，バックスラッシュ"\"に変換する。

戻 り 値――*path*を返す。

注　　意――_tobslash関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格――***Project LIBC Group***

関連項目――_toslash

_tolower

用　途——大文字を小文字に変換する。

書　式——#include <ctype.h>

```
int _tolower (int c);
```

解　説——_tolower マクロは *c* で指定された文字を小文字に変換する。これは tolower 関数とは異なり，変換は現在のロケールに影響されない。

戻り値——7 ビット ASCII 文字コードにしたがい，$c+$0x20 を返す。

注　意——*c* が大文字でなかった場合の動作は未定義である。また，_tolower はつねにマクロとして定義されるので，関数としての実体は存在しない。マクロに対する引数が副作用を伴わないように注意すること。

規　格——***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——tolower，_toupper，toupper

T

_toslash

用　　途——パス名に含まれるパス区切り記号をすべてスラッシュに変換する。

書　　式——#include <sys/xglob.h>

```
char *_toslash (char *path);
```

解　　説——_toslash 関数は *path* で指定されたパス名に含まれるすべてのパス区切り記号を，スラッシュ“/”に変換する。

戻 り 値——*path* を返す。

注　　意——_toslash 関数はライブラリの内部関数なので，移植性を考慮するならば使用しないこと。

規　　格——***Project LIBC Group***

関連項目——_tobslash

_toupper

用　途——小文字を大文字に変換する。

書　式——#include <ctype.h>

```
int _toupper (int c);
```

解　説——_toupper マクロは *c* で指定された文字を小文字に変換する。_toupper は toupper 関数とは異なり，変換は現在のロケールに影響されない。

戻り値——7 ビット ASCII 文字コードにしたがい，$c - \mathrm{0x20}$ を返す。

注　意——*c* が小文字でなかった場合の動作は未定義である。また，_toupper はつねにマクロとして定義されるので，関数としての実体は存在しない。マクロに対する引数が副作用を伴わないように注意すること。

規　格——***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——_tolower，tolower，toupper

T

tan

用　　途――正接 $tan(x)$ を求める。

書　　式――
```
#include <math.h>
double tan (double x);
#include <sys/xmath.h>
double _f_tan (double x);
double _fe_tan (double x);
double _fpu_tan (double x);
```

解　　説――tan 関数は x(単位：ラジアン) の正接 (タンジェント) を求める。各関数はそれぞれ次のように動作する。

- tan　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_tan　数値演算コプロセッサ命令を直接ドライブする
- _fe_tan　FLOAT パッケージを呼び出す
- _fpu_tan　I/O 数値演算コプロセッサを直接ドライブする

戻 り 値――正しく値が求められた場合，tan 関数は x の正接を返す。もし x が非数 (NaN) ならば変数 errno に EDOM を設定して同じく非数を返し，それ以外のエラーでは変数 errno に ERANGE を設定して計算結果を返す。ただし計算結果は不正確な値になる。

- EDOM　x の値が非数
- ERANGE　計算結果がオーバーフロー/アンダーフロー，または x の値が大きく計算結果の有効桁が一部失われる場合，または x の値が非常に大きくて計算結果の有効桁が完全に失われる場合

注　　意――<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは tan は _fe_tan の別名となり，__DIRECT_IOFPU__が定義されているときは tan は _fpu_tan の別名となる。また，__DIRECT_FPU__が定義されているときは tan は _f_tan の別名となる。

数値演算コプロセッサの場合のみ，ERANGE が設定される。

規　　格——*Project LIBC Group*, *ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *4.3BSD*, *SYSV*

関連項目——atan, isnan

tanh

用　途——双曲正接 $tanh(x)$ を求める。

書　式——

```
#include <math.h>
double tanh (double x);
#include <sys/xmath.h>
double _f_tanh (double x);
double _fe_tanh (double x);
double _fpu_tanh (double x);
```

解　説——tanh 関数は x の双曲正接（ハイパボリックタンジェント）を求める。各関数はそれぞれ次のように動作する。

- tanh　数値演算コプロセッサが使用できる場合は数値演算コプロセッサを直接ドライブし，使用できない場合は FLOAT パッケージを呼び出す
- _f_tanh　数値演算コプロセッサ命令を直接ドライブする
- _fe_tanh　FLOAT パッケージを呼び出す
- _fpu_tanh　I/O 数値演算コプロセッサを直接ドライブする

戻り値——正しく値が求められた場合は x の双曲正接を返す。もし x が非数（NaN）ならば，変数 errno に EDOM を設定して同じく非数を返す。

- EDOM　x の値が非数

注　意——<math.h>を読み込む前に，__DIRECT_FLOAT__が定義されているときは tanh は_fe_tanh の別名となり，__DIRECT_IOFPU__が定義されているときは tanh は_fpu_tanh の別名となる。また，__DIRECT_FPU__が定義されているときは tanh は_f_tanh の別名となる。

規　格——*Project LIBC Group*，*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——atan，isnan，tan

tell

用　途——ファイルポインタの位置を調べる。

書　式——
```
#include <unistd.h>
long tell (int fildes);
```

解　説——`tell` 関数は *fildes* で指定したファイルハンドルのファイルポインタの位置を求め，その値を返す。

戻り値——正常に取得できた場合は 0 を返し，失敗した場合には−1 を返して，変数 `errno` にその原因を示すエラーコードを設定する。

- `EBADF`　不正な *fildes* を指定した

注　意——ここでいうファイルポインタとは，ファイルハンドルを通して**低水準入出力**におけるファイルポインタのことであり，`stdio` **ライブラリ**のファイルポインタと混同しないように注意すること。

規　格——*4.3BSD*，*SYSV*

関連項目——`fgetpos`，`fseek`，`fsetpos`，`lseek`

telldir

用　途――ディレクトリストリームのポインタ位置を返す。

書　式――#include <dirent.h>

```
long telldir (DIR *dirp);
```

解　説――telldir 関数は，*dirp* が指すディレクトリストリームの現在のポインタ位置を返す。

戻り値――正常に位置を求められた場合はその位置を返し，失敗した場合には–1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EPERM　不正なディレクトリストリームを指定した

規　格――***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――closedir，readdir，rewinddir，seekdir，telldir

tempnam

用　途――テンポラリファイル名を作成する。

書　式――#include <stdio.h>

```
char *tempnam (const char *dir, const char *prefix);
```

解　説――tempnam 関数は *dir* と *prefix* をもとにして，テンポラリファイル名を作成する。*dir* はテンポラリファイルを作成するディレクトリを指定し，*prefix* にはアプリケーション固有のプレフィクス文字 (先頭 5 文字まで使用される) を指定する。もし *dir* が NULL だったり，*dir* に指定したディレクトリが存在しなかったりして書き込みできない場合は，代わりに P_tmpdir が使用される。tempnam 関数はこれらを使ってテンポラリファイルのテンプレートを作成し，最終的には mktemp 関数を用いて実際のテンポラリファイル名を得る。

戻り値――正常にテンポラリファイル名を作成できた場合はそれを格納する領域を確保し，そこに結果を格納してからそのポインタを返す。失敗した場合は NULL を返し，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　意――tempnam 関数が返したポインタが指す領域は，ユーザが責任をもって解放しなければならない。また，tempnam 関数はファイル名を作成するだけであり，実際にファイルは作成されない。

規　格――*XPG3*， *AES/OS*， *SYSV*

関連項目――creat，fopen，free，malloc，mktemp，open，tmpfile，tmpnam

time

用　　途——現在時刻を取得する。

書　　式——#include <time.h>

```
time_t time (time_t *tp);
```

解　　説——time 関数は，1970年1月1日 (エポックタイム) から今現在までの経過秒数 (暦時間) を求めてその値を返すとともに，もしも *tp* が NULL でないなら *tp* が指す領域にもその結果を格納する。

戻 り 値——正常に求めることができた場合は今現在までの経過秒数を，失敗した場合は−1を返す。

規　　格——*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——clock，difftime，mktime

tmpfile

用　　途――テンポラリファイルを作成する。

書　　式――#include <stdio.h>

```
FILE *tmpfile (void);
```

解　　説――tmpfile 関数はテンポラリファイルを作成し，それに対応する**ファイルストリーム**を**更新モード** "w+" で割り当てる。なお，このテンポラリファイルは fclose 関数によってクローズされると，自動的に削除される。

戻 り 値――正常に作成できた場合はそのストリームへのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- EMFILE　これ以上ファイルをオープンできない
- ENOSPC　ディスクがいっぱいである
- ENOMEM　メモリが足りなくなった

互 換 性――テンポラリファイルは，デフォルトでは**テキストモード**でオープンされる。設定を変更したい場合は，あらかじめ fmode 関数でデフォルトの変換モードを設定しておかなければならない。

規　　格――*ANSI C*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

関連項目――fmode，fopen，tempnam，tmpnam，unlink

変　　更――従来ではテンポラリファイルをテキストモードでオープンしていたが，現在の *LIBC* ではバイナリモードでオープンされる。

tmpnam

用　　途――テンポラリファイルのファイル名を作成する。

書　　式――#include <stdio.h>

```
char *tmpnam (char *s);
```

解　　説――tmpnam関数はテンポラリファイルのファイル名を作成し，その結果を*s*が指す領域に格納する。ファイル名はパス名として有効であり，すでに存在しているファイル名とは決して一致しない。

tmpnam関数は同じプロセスから呼び出された場合，最低TMP_MAX回は異なるファイル名を作成することができるが，それを越えた場合には，必ずしもユニークになるとは限らない（しかし現実的には，重なることはない）。なお，TMP_MAXは<stdio.h>に定義されている。

戻 り 値――正常にファイル名を作成できた場合，*s*がNULLでなければ*s*が指す領域に，NULLならば関数内部の静的領域にそれぞれ格納され，その領域へのポインタを返す。もし失敗した場合はNULLを返し，変数errnoにその原因を示すエラーコードを設定する。

- EINVAL　ユニークなファイル名の作成に失敗した

注　　意――関数内部の静的領域を使用する場合，その結果は関数を呼び出すたびに上書きされることに注意すること。また*s*にNULLでない値を指定する場合には，*s*は最低でもL_tmpnamバイトの領域を指していなければならない。なお，L_tmpnamは<stdio.h>に定義されている。

規　　格――*ANSI C*, *POSIX.1*, *XPG3*, *AES/OS*, *SYSV*

関連項目――fopen, open, tempnam, tmpfile, unlink

toascii

用　途——7 ビット ASCII 文字に変換する。

書　式——`#include <ctype.h>`
`int toascii (int c);`

解　説——toascii 関数は *c* を 7 ビット ASCII 文字コードに変換する。

戻り値——*c* & 127 を返す。

注　意——通常，toascii 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。toascii 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

規　格——***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isascii

toiso

用　　途――8ビットISO文字に変換する。

書　　式――
```
#include <ctype.h>
int toiso (int c);
```

解　　説――toiso関数はcを8ビットISO文字コードに変換する。

戻 り 値――c & 255を返す。

注　　意――通常，toiso関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。toiso関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

規　　格――*Project LIBC Group*

関連項目――isascii

tolower

用　　途——大文字を小文字に変換する。

書　　式——#include <ctype.h>

```
int tolower (int c);
```

解　　説——tolower 関数は *c* で指定された文字を，現在のロケールの LC_CTYPE カテゴリにしたがって大文字から小文字に変換する。ただし，変換は大文字に対してのみ行われ，それ以外の文字は変化しない。

戻 り 値——*c* を小文字に変換した文字を返す。

注　　意——通常，tolower 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。tolower 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格——***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——_tolower，_toupper，toupper

T

toupper

用　　途――小文字を大文字に変換する。

書　　式――#include <ctype.h>

```
int toupper (int c);
```

解　　説――toupper 関数は *c* で指定された文字を，現在のロケールの LC_CTYPE カテゴリにしたがって小文字から大文字に変換する。ただし，変換は小文字に対してのみ行われ，それ以外の文字は変化しない。

戻 り 値――*c* を大文字に変換した文字を返す。

注　　意――通常，toupper 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。toupper 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――_tolower，tolower _toupper，

truncate

用　途——ファイルの長さを変更する。

書　式——

```
#include <unistd.h>
int truncate (const char *path, off_t len);
```

解　説——truncate 関数は *path* で指定したファイルの長さを，*len* バイトに変更する。もし *len* が現在のファイル長よりも短い場合は，その間のデータはすべて捨てられる。しかし反対に長い場合は，その間のデータはすべて 0 で埋められる。

ただし，truncate 関数はキャラクタデバイスに対しては動作しない。またファイルは書き込み可能であること。

戻り値——正常に変更できた場合は 0 を返し，失敗した場合には−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOENT　*path* で指定したファイルが存在しない
- EROFS　リードオンリーファイルシステムである
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

規　格——*AES/OS*

関連項目——chsize, creat, ftruncate, open

ttyname

用　　途——端末のデバイス名を調べる。

書　　式——#include <unistd.h>

```
char *ttyname (int fildes);
```

解　　説——ttyname関数は，*fildes*で指定したファイルハンドルが指すファイルが端末デバイス(キャラクタデバイス)であるかどうかを調べ，もし端末デバイスであればそのデバイス名を返す。

戻 り 値——端末デバイスならばその名前へのポインタを返し，端末デバイスでなければNULLを返して，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　不正な*fildes*を指定した
- ENOTTY　端末デバイスではない

注　　意——結果は関数内部の静的領域に格納されるため，関数を呼び出すたびに内容が上書きされることに注意すること。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——isatty

tzset

用　途――タイムゾーン情報 (地域時間情報) を初期化する。

書　式――
```
#include <time.h>
void tzset (void);
```

解　説――tzset 関数は環境変数 TZ を参照して，**協定世界時** (UTC) と**地域時間**を相互に変換するための地域時間情報を初期化し，地域時間名を示す変数 tzname，時差を示す変数 timezone，夏時間を示す変数 daylight を設定する。

変数 tzname は 2 つの要素をもち，tzname[0] が通常の地域時間名，tzname[1] が夏時間の期間中の地域時間名を表す。

変数 daylight は**夏時間**の補正を行うかどうかを表している。もし変数 daylight が 0 ならば夏時間の期間中ではなく (補正は行われない)，0 以外の値ならば夏時間の期間中 (補正が行われる) である。

変数 timezone は地域時間と協定世界時の時差を表しており，時差±0 の地域から西回りに計算した時差の秒数である。時差が+9 時間の日本一帯の地域であれば，変数 timezone は−32400 になる。時差とは符合が逆になることに注意すること。

環境変数 TZ の設定フォーマットは ***POSIX.1*** に規定されたものを採用しており，

```
TZ=stdoffset[dst[offset][start[/time],end[/time]]]
```

のようなフォーマットである。左から順に，

- std　地域時間の地域時間帯名称 (3 文字またはそれ以上)
- offset　通常の地域時間から協定世界時への補正値で，形式は [sign]hh[:mm[:ss]]
- dst　地域時間の夏時間の期間中の地域時間帯名称 (3 文字またはそれ以上)
- offset　夏時間の期間中の地域時間から協定世界時への補正値で，形式は [sign]hh[:mm[:ss]]
- start[/time]　夏時間の開始日時
- end[/time]　夏時間の終了日時

を表している。このうち，夏時間の開始/終了日時のフォーマットは次の 2 通りである。

- [J]n[/hh:mm:ss]　J で始まる場合は Julian 日付とし，閏日はカウントされない ($1 \leq n \leq 365$) 。つまり 2 月 28 日が

T

$n = 59$ で，3月1日が $n = 60$ となる。Jで始まらない場合は0を底とするJulian日付とする。閏日はカウントされる ($0 \le n \le 365$) ので，2月29日は $n = 60$ となり3月1日は $n = 61$ となる

- Mm.n.d[/hh:mm:dd]　Mで始まる場合は m 月の第 n 週の第 d 日となる。ここで各値の範囲は，$0 \le d \le 6$, $1 \le n \le 5$, $1 \le m \le 12$ であり，n が5のときは「その月の最後の週の最終日」を表す。また，n が1のときは「その月の第 d 日目」を表す。$d = 0$ は日曜日を表す

夏時間の開始/終了時間を指定する /time 部分は offset フィールドと同じだが，符合は識別しない。指定しない場合は 02:00:00 と解釈される。

便宜上，std と dst のフィールドは‘:’，‘,’，‘-’，‘+’または数字を検出した時点で完結することにする。つまり，TZ=JST:-9 や TZ=JST,-9 といった指定が可能となる。

最後に具体例を示す。

```
TZ=JST-9USA+5:30:25:J320/02:00:00,M12.5.6/02:00:00
```

上記の例は，地方時間が JST で協定世界時と -9 時間の差があり，夏時間の期間中は時間帯名称が USA に変更し，協定世界時と $+5$ 時間 30 分 25 秒の差があることを示す。また夏時間は，第 320 日目の 2:00:00 から 12 月第 5 週の土曜日 2:00:00 までである。なお念のため補足するが，このような例は実際には存在しない。

戻 り 値——なし。

注　　意——環境変数 TZ が設定されない場合，既定値として次の値を設定する。

```
tzname[0] = "JST";
tzname[1] = "   ";
daylight  = 0;
timezone  = -32400;
```

互 換 性——標準のライブラリを利用すると，簡易型の tzset 関数がリンクされる。これは，環境変数 TZ を見ず，つねに既定値で設定するものである。もしも，ここで解説したフルスペックの tzset 関数を使用したければ，明示的に “libtz.a” をリンクしなければならない。

規　格——*Project LIBC Group*, *POSIX.1*, *XPG3*, *AES/OS*, *SYSV*

関連項目——ctime, localtime, mktime, strftime

バ　グ——**Human68k** のシステムコールでは地域時間しか取得することができないため，この地域時間から見かけ上の協定世界時を求める手順を，日本でしか通用しない方法でハードコーディングしている。したがって，日本および日本が属する時間帯以外で使用すると，時間が狂ってしまう。

T

umask

用　　途——ファイルモードの新規作成用マスクを設定する。

書　　式——

```
#include <sys/stat.h>
mode_t umask (mode_t cmask);
```

解　　説——umask 関数は，*cmask* で指定したファイルマスクをカレントファイルマスクとして設定する。ここで指定したファイルマスクは，creat 関数や open 関数，mkdir 関数などで新規にファイル，ディレクトリを作成する場合に，それぞれの関数に引数として与えられたモードに適用される。デフォルトでは0である。

戻り値——以前のカレントファイルマスクを返す。

規　　格——*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——creat，mkdir，open

uname

用　途──システムに関する情報を取り出す。

書　式──#include <sys/utsname.h>

```
int uname (struct utsname *name);
```

解　説──uname 関数は現在のシステム，および**オペレーティングシステム**に関する情報を取得し，*name* が指す領域に格納する。utsname 構造体は次のとおり。

```
struct utsname {
    char *sysname;  /* オペレーティングシステムの名前 */
    char *nodename; /* ネットワーク上のノード名（ホスト名） */
    char *release;  /* オペレーティングシステムのリリース番号 */
    char *version;  /* オペレーティングシステムのバージョン番号 */
    char *machine;  /* マシンのハードウェア名 */
};
```

上記の項目はそれぞれ次のように求めている。

- sysname　　オペレーティングシステムの名前としてつねに "Human68k" を返す
- nodename　　環境変数 HOST が設定されていればその文字列を，設定されていなければ "unknown" を返す
- release　　**Human68k** のメジャーバージョン，たとえば Human2.03 ならば "2" を返す
- version　　**Human68k** のマイナーバージョン，たとえば Human2.03 ならば "3" を返す
- machine　　ライブラリを実行中のマシンが **X68000** ならば "X68000" を，**X68030** ならば "X68030" を返す

戻り値──正常に取得できた場合は 0 を返す。エラーはない。

規　格──*Project LIBC Group*，*POSIX.1*，*XPG3*，*AES/OS*，*SYSV*

U

ungetc

用　　途――入力ファイルストリームにデータを押し戻す。

書　　式――#include <stdio.h>

```
int ungetc (int c, FILE *stream);
```

解　　説――ungetc 関数は，*stream*で指定された入力用のファイルストリームに *c*を unsigned char 型に変換してから押し戻す。ただし *c*に，EOF を指定した場合は何も行わない。

押し戻されたデータは後に続く読み出し操作で最優先に取り出されることになるが，あくまでもバッファ上の操作であり，ファイルの実体は一切変更されない。また押し戻しが行われた後は，ストリームに設定されていた終端指示子はクリアされる。

戻り値――正常に押し戻せた場合は *c*を返し，失敗した場合は EOF を返す。

注　　意――ungetc 関数は最低限1バイト(*LIBC*では1幅広文字)を押し戻せることを保証しているが，それ以上は状況によっては押し戻せないことがあるので注意すること。

規　　格――***ANSI C***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――fseek，fsetpos，getc，rewind，setbuf

ungetch

用　　途――コンソールにデータを押し戻す。

書　　式――#include <conio.h>

```
int ungetch (int c);
```

解　　説――ungetch 関数はコンソールに対して *c* で指定した文字を押し戻し，その文字コードを返す。押し戻された文字は，後で getch 関数，getche 関数，cgets 関数などで読み出しが行われた場合に，再度入力データとして取り出されることになる。ungetch 関数はつねにコンソールと直接データをやりとりするので，標準入力の状態に左右されることも stdio ライブラリによるバッファリングの作用も受けない。

戻 り 値――*c* を返す。エラーはない。

注　　意――コンソール用の押し戻しバッファは 1 バイト分しかない。続けて 2 回以上 ungetch 関数を呼び出した場合，データは失われてしまうことになるので注意すること。

規　　格――***Project LIBC Group***，　***MS–C 7.0***

関連項目――cgets，getch，getche，ungetc

U

unlink

用　途――ファイルを削除する。

書　式――
```
#include <unistd.h>
int unlink (const char *path);
```

解　説――unlink関数は，*path*で指定したファイルを削除する。*path*には，現在creat関数やopen関数でオープンしているファイルも指定することができるが，その場合実際に削除されるのはclose関数でファイルがクローズされたときであり，unlink関数は削除の予約だけを行う。

戻り値――正常に削除できた場合は0を返し，失敗した場合には–1を返して，変数errnoにその原因を示すエラーコードを設定する。

- ENOENT　*path*で指定したファイルが存在しない
- EISDIR　ディレクトリである
- EROFS　リードオンリーファイルシステムである
- ELOOP　シンボリックリンクのネストが深すぎるか，またはループしている

注　意――close関数とunlink関数は**UNIX**上のunlinkシステムコールをシミュレートしているが，**Human68k**のファイルシステムではリンクカウントを保持できないため，つねにカウントを1として扱っている。

また，その実現方法は完全ではないので，理論的には削除予約したはずのファイルが正しく削除されずに残ってしまうこともあり得る。したがってunlink関数は，ファイルをクローズした後に実行するのが望ましいといえる。

unlink関数は*path*で指定したファイルが現在オープンされているかどうかを調べるが，それは*LIBC*内部のテーブルを参照しているにすぎない。したがって，DOSコールなどを用いて直接オープンしているファイルに対しては有効ではない。そのようなファイルをオープンしている最中に削除すると，**Human68k**の仕様により管理領域が破壊される。

規　格――***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目――close，remove，rmdir

変　更――従来では*path*にディレクトリを指定するとエラーとなっていたが，現在の*LIBC*ではディレクトリを削除するようにした。

usleep

用　途——マイクロ秒単位のスリープを実行する。

書　式——#include <unistd.h>

```
unsigned int usleep (unsigned int useconds);
```

解　説——usleep 関数は，*useconds* で指定したマイクロ秒数だけ現在実行中のプロセスをスリープさせる。プロセスは指定した時間スリープするが，シグナルが通知されてシグナルハンドラが呼び出された場合はスリープ状態から復帰する。

戻り値——指定した時間が経過した場合は 0 を，途中でシグナルによって中断された場合は残りスリープ時間 (マイクロ秒単位) を返す。

注　意——引数はマイクロ秒単位で指定するが，**Human68k** では $1/_{100}$ 秒までの精度しかない。

規　格——***Project LIBC Group***，***4.3BSD***，***SYSV***

関連項目——pause，signal，sleep

U

utime

用　　途——ファイルのタイムスタンプを変更する。

書　　式——#include <utime.h>

```
int utime (const char *name, struct utimbuf *times);
```

解　　説——utime 関数は *name* が指すファイルのタイムスタンプ (最終変更日付および最終変更時刻) を，*times* が NULL であれば現時刻に変更し，NULL でなければ *times* が指す構造体の時間に変更する。utimebuf 構造体は次のとおり。

```
struct utimbuf {
    time_t actime;  /* 最終アクセス時刻 */
    time_t modtime; /* 最終更新時刻 */
};
```

また構造体メンバは，1970 年 1 月 1 日 (エポックタイム) から指定時刻までの経過秒数で指定する。

戻 り 値——正常にタイムスタンプが変更できた場合は 0 を返し，失敗した場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　不正な時間を指定した
- ENOENT　*name* で指定したファイルが存在しない
- ENOENT　書き込み禁止ファイルを指定した

注　　意——**Human68k** では actime は無視され，modtime のみが参照される。またディスクには UTC ではなく，JST(日本標準時) で書き込まれる。

規　　格——***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——localtime，time

va_arg

用　途──可変長引数リストから引数を1つ取り出す。

書　式──

```
#include <stdarg.h>
type va_arg (va_list ap, type);
#include <varargs.h>
type va_arg (va_list ap, type);
```

解　説──va_argは，*ap*で指定した**可変長引数リスト**から*type*で指定した変数型の引数を1つ取り出し，その値を返す。このとき*ap*は，データの**アラインメント**や引数の**スタック**への積み方を考慮して，適切な方法で引数リストにアクセスする。

なお引数を取り出した後，*ap*のポインタは次の引数を取り出すことができるように進められる。

戻り値──引数リストから*type*で指定した変数型のデータを取り出し，その値を返す。同様に，戻り値の変数型も*type*型となる。

注　意──*ap*は使用する前に，必ずva_startで初期化されている必要がある。また，*ap*の値や可変長引数リストの内容が適切でなかった場合，va_argの戻り値は不定である。

<varargs.h>を使用する場合，関数宣言は都合上，***K&R***スタイルを用いなければならない。***ANSI C***スタイルを用いたい場合は，必ず<stdarg.h>のほうを用いること。

互換性──<varargs.h>は，古いタイプのソースファイルとの互換性のためにだけ用意されている。通常は，***ANSI C***規格の<stdarg.h>に定義されているインタフェイスを用いること。

また<varargs.h>と<stdarg.h>は，決して同時に用いてはならない。

規　格──***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目──va_end，va_start

サンプル――List 1-5 ● <stdarg.h>の使用例

```
/*
** 引数に積まれたn個の文字列をすべて表示する
** output (int n, str1, str2, ...., strn);
*/

#include <stdio.h>
#include <stdarg.h>

void output (int n, ...) /* 引数の最後は ... と書く */
{
    va_list ap;           /* 可変長引数リストへのポインタ */
    char *p;

    /* 引数リストへのアクセスを開始 */
    va_start (ap, n);

    /* 引数リストから char * のデータを取り出す */
    while ((p = va_arg (ap, char *)) != (char *) 0)
        printf ("string = %s\n", p);

    /* 引数リストへのアクセスを終わる */
    va_end (ap);
}
```

サンプル――List 1-6 ● <varargs.h>の使用例

```
/*
** 引数に積まれたn個の文字列をすべて表示する
** output (int n, str1, str2, ...., strn);
*/

#include <stdio.h>
#include <varargs.h>

void output (n, va_alist) /* 引数の最後に va_alist を書く */
    int n;                /* varargs.h を使う場合は K&R スタイル */
    va_dcl                /* va_alist のための宣言。セミコロンはつけない */
{
    va_list ap;           /* 可変長引数リストへのポインタ */
    char *p;

    /* 引数リストへのアクセスを開始 */
    va_start (ap);

    /* 引数リストから char * のデータを取り出す */
    while ((p = va_arg (ap, char *)) != (char *) 0)
        printf ("string = %s\n", p);

    /* 引数リストへのアクセスを終わる */
    va_end (ap);
}
```

va_end

用　途——可変長引数リストへのアクセスを終了する。

書　式——
```
#include <stdarg.h>
void va_end (va_list ap);
#include <varargs.h>
void va_end (va_list ap);
```

解　説——va_end は，*ap*で指定した**可変長引数リスト**へのアクセスを終わりにし，適切な事後処理をする。可変長引数リストへのアクセス方法など，詳細は va_arg を参照のこと。

戻り値——なし。

注　意——<varargs.h>を使用する場合，関数宣言は都合上，***K&R***スタイルを用いなければならない。***ANSI C***スタイルを用いたい場合は，必ず<stdarg.h>のほうを用いること。

互換性——<varargs.h>は，古いタイプのソースファイルとの互換性のためにだけ用意されている。通常は，*ANSI C*規格の<stdarg.h>に定義されているインタフェイスを用いること。

また<varargs.h>と<stdarg.h>は，決して同時に用いてはならない。

規　格——*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——va_arg，va_start

V

va_start

用　　途——可変長引数リストへのアクセスを開始する。

書　　式——
```
#include <stdarg.h>
void va_start (va_list ap, parmn);
#include <varargs.h>
void va_start (va_list ap);
```

解　　説——<stdarg.h>を使用する場合，va_startは*parmn*で指定した引数の次からが可変長引数リストであるとして*ap*を初期化し，引数リストへのアクセスができるようにする。

また<varargs.h>を使用する場合，va_startは関数宣言でva_alistを指定した位置からが可変長引数リストであるとして*ap*を初期化し，引数リストへのアクセスができるようにする。

どちらも初期化された後は，va_argによって求める引数を取り出すことができる。可変長引数リストへのアクセス方法など，詳細はva_argを参照のこと。

戻り値——なし。

注　　意——<varargs.h>を使用する場合，関数宣言は都合上，***K&R***スタイルを用いなければならない。***ANSI C***スタイルを用いたい場合は，必ず<stdarg.h>のほうを用いること。

互換性——<varargs.h>は，古いタイプのソースファイルとの互換性のためにだけ用意されている。通常は，***ANSI C***規格の<stdarg.h>に定義されているインタフェイスを用いること。

また<varargs.h>と<stdarg.h>は，決して同時に用いてはならない。

規　　格——***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——va_arg，va_end

vfprintf

用　　途――可変長引数リストをフォーマット出力する。

書　　式――#include <stdio.h>

```
int vfprintf (FILE *stream, const char *format, va_list ap);
```

解　　説――vfprintf 関数は *format* で指定した表示フォーマットにしたがって，*ap* で指定した**可変長引数リスト**のデータを *stream* のファイルストリームに出力する。表示フォーマットや機能的な説明については，printf 関数を参照のこと。

戻 り 値――正常に出力できた場合はそのバイト数を返し，何らかのエラーによって失敗した場合はストリームの**エラー指示子**を設定して負の値を返し，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　　意――*ap* は，あらかじめ<stdarg.h>や<varargs.h>で定義されているマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　　格――*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目――fprintf，printf，vfprintf，vprintf

V

vfscanf

用　途——ファイルストリームから可変長引数リストを用いてフォーマット入力を実行する。

書　式——
```
#include <stdio.h>
int vfscanf (FILE *stream, const char *format, va_list ap);
```

解　説——vfscanf 関数は *format* で指定したフォーマット文字列にしたがって，必要なデータ列を *stream* のファイルストリームから入力し，その結果を *ap* の指す領域に格納する。ただし vscanf 関数では，この引数は**可変長引数リスト**から取り出される。

入力フォーマット文字列の指定の仕方や詳細な説明については，fscanf 関数を参照のこと。

戻り値——正常に入力できた場合は実際に入力できた項目数を返し，失敗した場合は EOF を返してストリームのエラー指示子を設定し，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——引数に指定するポインタは，結果を格納するだけの十分な大きさの領域を指していなければならない。

また *ap* は，あらかじめ<stdarg.h>や<varargs.h>で定義されてるマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　格——*Project LIBC Group*

関連項目——fscanf，scanf，sscanf，vscanf，vsscanf

vprintf

用　途——可変長引数リストをフォーマット出力する。

書　式——#include <stdio.h>

```
int vprintf (const char *format, va_list ap);
```

解　説——vprintf 関数は *format* で指定した表示フォーマットにしたがって，*ap* で指定した可変長引数リストのデータを標準出力ストリームに出力する。表示フォーマットや機能的な説明については，printf 関数を参照のこと。

戻り値——正常に出力できた場合はそのバイト数を返し，何らかのエラーによって失敗した場合は標準出力ストリームのエラー指示子を設定して負の値を返し，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクがいっぱいである

注　意——*ap* は，あらかじめ<stdarg.h>や<varargs.h>で定義されているマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　格——*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目——printf，vfprintf，vsprintf

V

vscanf

用　　途――標準入力ファイルストリームから可変長引数リストを用いてフォーマット入力を実行する。

書　　式――#include <stdio.h>

```
int vscanf (const char *format, va_list ap);
```

解　　説――vscanf 関数は *format* で指定したフォーマット文字列にしたがって，必要なデータ列を標準入力に割り当てられたファイルストリームから入力し，その結果を *ap* の指す領域に格納する。ただし vscanf 関数では，この引数は可変長引数リストから取り出される。

入力フォーマット文字列の指定の仕方や詳細な説明については，fscanf 関数を参照のこと。

戻 り 値――正常に入力できた場合は実際に入力した項目数を返し，失敗した場合は EOF を返してストリームのエラー指示子を設定し，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　　意――引数に指定するポインタは，結果を格納するだけの十分な大きさの領域を指していなければならない。

また *ap* は，あらかじめ<stdarg.h>や<varargs.h>で定義されているマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　　格――***Project LIBC Group***

関連項目――fscanf，scanf，sscanf，vfscanf，vsscanf

vsprintf

用　途——可変長引数リストをフォーマット出力する。

書　式——#include <stdio.h>

```
int vsprintf (char *s, const char *format, va_list ap);
```

解　説——vsprintf 関数は *format* で指定した表示フォーマットにしたがって，*ap* で指定した可変長引数リストのデータを文字列に変換し，その結果を *s* の指す領域に格納する。表示フォーマットや機能的な説明については，printf 関数を参照のこと。

戻り値——変換したバイト数 (終端の null 文字は含まない) を返す。

注　意——*s* は結果を格納するだけの十分な大きさの領域を指していなければならない。

また *ap* は，あらかじめ<stdarg.h>や<varargs.h>で定義されているマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　格——*ANSI C*，*XPG3*，*AES/OS*，*SYSV*

関連項目——printf，sprintf，vfprintf，vprintf

V

vsscanf

用　　途――文字列から可変長引数リストを用いてフォーマット入力を実行する。

書　　式――#include <stdio.h>

```
int vsscanf (const char *s, const char *format, va_list ap);
```

解　　説――vsscanf 関数は*format*で指定したフォーマット文字列にしたがって，必要なデータ列を *s*で指定した文字列から入力し，その結果を *ap*の指す領域に格納する。ただし vsscanf 関数では，この引数は可変長引数リストから取り出される。

入力フォーマット文字列の指定の仕方や詳細な説明については，fscanf 関数を参照のこと。

戻 り 値――正常に入力できた場合は実際に入力できた項目数を返し，失敗した場合は EOF を返す。

注　　意――引数に指定するポインタは，結果を格納するだけの十分な大きさの領域を指していなければならない。

また *ap*は，あらかじめ<stdarg.h>や<varargs.h>で定義されているマクロ va_start，va_arg，va_end で初期化されていなくてはならない。

規　　格――***Project LIBC Group***

関連項目――fscanf，scanf，sscanf，vfscanf，vscanf

wabs

用　途――short 型の値の絶対値を取得する。

書　式――#include <stdlib.h>

```
short wabs (short n);
```

解　説――wabs 関数は *n* の絶対値を返す。

戻り値――*n* の short 型絶対値を返す。

注　意――通常，wabs 関数はマクロとして定義されるが，__NO_STDLIB_INLINE__ が定義された場合には実体をもつ関数となる。

規　格――*Project LIBC Group*

関連項目――abs，div，labs，ldiv

W

wcstombs

用　　途――幅広文字列をマルチバイト文字列に変換する。

書　　式――#include <stdlib.h>

```
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

解　　説――wcstombs 関数は，*pwcs* で指定された幅広文字列をマルチバイト文字列に変換し，結果を *s* が指す領域に格納する。変換は null 文字に到達するか，*n* バイトを格納するまで行われる。

また，変換はロケールの LC_CTYPE カテゴリに影響される。もし現在のロケールが状態依存体系の文字コードを用いていた場合，変換結果はシフト状態を初期化してから格納される。

戻 り 値――変換して格納されたバイト数 (終端の null 文字は含まない) を返す。ただし途中で不正な幅広文字に出会った場合，変換はそこで中止され，−1 を返して変数 errno にその原因を示すエラーコードを設定する。

- EINVAL　変換中に不正な幅広文字に出会った

注　　意――*LIBC* は C ロケールしかサポートしていない。

規　　格――***Project LIBC Group***，***ANSI C***，***AES/OS***，***SYSV***

関連事項――mblen，mbstowcs，mbtowc，wctomb

wctomb

用　　途――幅広文字をマルチバイト文字に変換する。

書　　式――

```
#include <stdlib.h>
int wctomb (char *s, const wchar_t wchar);
```

解　　説――wctomb 関数は，*wchar* で指定された幅広文字をマルチバイト文字に変換し，*s* が指す領域に格納する。

変換はロケールの LC_CTYPE カテゴリに影響される。もし現在のロケールが状態依存体系の文字コードを用いていた場合，*wchar* に null 文字を指定することで，シフト状態を初期化することができる。

戻 り 値――*wchar* が正しい幅広文字だった場合は，変換したマルチバイト文字を構成するバイト数を返し，不正な幅広文字だった場合は−1 を返して，変数 errno にその原因を示すエラーコードを設定する。

また *s* に NULL を指定した場合，現在のロケールの文字コードが状態依存体系であれば 0 以外の値を，状態依存体系でなければ 0 を返す。

- EINVAL　不正な幅広文字に出会った

注　　意――*LIBC* は C ロケールしかサポートしていない。

規　　格――***Project LIBC Group*，*ANSI C*，*AES/OS*，*SYSV***

関連項目――mblen，mbstowcs，mbtowc，wcstombs

W

write

用　途——ファイルへ書き込む。

書　式——

```
#include <unistd.h>
int write (int fildes, const char *buff, unsigned int n);
```

解　説——write関数は，*fildes*で指定したファイルハンドルが指すファイルに対して*buff*が指す領域から*n*バイトを書き込む。書き込み中に何らかのエラーが発生すると，write関数はその時点で処理を中断する。その結果，ファイルポインタは書き込んだバイト数分だけ進む。ただし，*n*が0だったときは何も行わない。

ファイルがバイナリモードでオープンされていた場合，データはすべてそのままファイルに書き込まれるが，テキストモードでオープンされていた場合は，データ中に含まれるすべてのLFをCRLFに変換してから書き込まれる。その結果，正常に処理できても戻り値が*n*を越えることがある。

戻り値——正常に書き込めた場合は，実際に書き込んだバイト数を返す。何らかのエラーによって失敗した場合には−1を返して，変数errnoにその原因を示すエラーコードを設定する。

*n*が0だった場合は0を返す。

- EBADF　不正な*fildes*を指定した

規　格——***Project LIBC Group***，***POSIX.1***，***XPG3***，***AES/OS***，***4.3BSD***，***SYSV***

関連項目——fcntl，lseek，open

変　更——従来では，テキストモードでのwrite関数の戻り値はread関数と同じくLF→CR，LF変換の結果によって変化していたが，現在の***LIBC***では，この変換に関係なく指定されたバッファのうち何バイトを書き込めたかを返すようになった。

従来では***XC***，***MS-C 7.0***との互換性に難点があったが，これらの変更により現在の***LIBC***では完全に互換性がある。

Chapter 2
DOS コールライブラリ

本章には Human68k の DOS コールライブラリのマニュアルを掲載します。*LIBC*では従来の *XC*との互換性を考慮し，名前が異なる以外は基本的な動作はすべて同じに作成されています。

本書では，DOS コールについては関数ごとのマニュアルにとどめ，各デバイスなどの詳しい説明はしません。詳しく知りたい場合は，他の書籍などを参照してください。

_dos_allclose

用　途——オープンしているすべてのファイルをクローズする。

書　式——
```
#include <sys/dos.h>
void _dos_allclose (void);
```

解　説——_dos_allclose 関数は現在オープンされているファイルをすべてクローズする。**子プロセス**がクローズし忘れたファイルについても同様にクローズするが，自プロセスの**親プロセス**(自プロセスを呼び出したプロセス) がオープンしたファイルについてはクローズしない。

_dos_allclose 関数は DOS コール 0xFF1F を発行することによって処理される。

戻り値——なし。

注　意——_dos_allclose 関数は **Human68k** レベルでファイルをクローズしている。したがって，stdio ライブラリ (fopen 関数など) でファイルをオープンしている場合は使用してはならない。

規　格——*XC*

_dos_assign

用　途——仮想ドライブ/仮想ディレクトリの割り当てリストの取得/作成/解除を行う。

書　式——
```
#include <sys/dos.h>
int _dos_getassign (const char *buff1, char *buff2);
int _dos_makeassign (const char *buff1,
                     const char *buff2, int mode);
int _dos_rassign (char *buff);
```

解　説——_dos_getassign 関数は，仮想ドライブ，仮想ディレクトリの割り当て状況を調べる。

戻り値により，*buff2* に次に示す割り当て状況がセットされる。

- 0x40　割り当てが存在しないことを示す。*buff1* で指定したドライブのカレントディレクトリを *buff2* へセットする
- 0x50　*buff1* をアクセスしたとき，実際にアクセスされるディレクトリが *buff2* へセットされる
- 0x60　*buff1* を実際にアクセスしようとしたとき，アクセスしなくてはならないディレクトリが *buff2* へセットされる

buff1 に_dos_makeassign 関数での 1 番目の引数と同じ内容を指定すると，2 番目の引数で指定した内容が *buff2* にセットされる。

_dos_makeassign 関数は仮想ドライブ/仮想ディレクトリの割り当てを行う。

mode に指定する値は次のとおり。

- 0x50　仮想ドライブの割り当てを行う。*buff1* のドライブをアクセスしたとき，実際には *buff2* のディレクトリがアクセスされる
- 0x60　仮想ディレクトリの割り当てを行う。*buff1* のディレクトリをアクセスしたとき，*buff2* のドライブがアクセスされる

_dos_rassign 関数は仮想ドライブ/仮想ディレクトリの割り当てを解除する。*buff* には_dos_makeassign 関数の第 1 引数と同じ内容を指定すること。

これら 3 つの関数は DOS コール 0xFF5F(0xFF8F) を発行することによって処理される。

戻り値——_dos_getassign関数は，成功した場合は次に示す割り当てモードを返し，失敗した場合は負の値を返す。

- 0x40　割り当てが存在しない
- 0x50　仮想ドライブの割り当て
- 0x60　仮想ディレクトリの割り当て

_dos_makeassign関数と_dos_rassign関数は，失敗した場合に負の値を返す。

互換性——**Human68k** ver.3ではDOSコールの番号が変更(+0x30)されたことに注意すること。

規　格——*XC*

_dos_breakck

用　途――ブレークチェックを設定する。

書　式――

```
#include <sys/dos.h>
int _dos_breakck (int flag);
```

解　説――_dos_breakck 関数はブレークチェックの設定を行う。

flag に設定可能な値は次のとおり。

- 0　特定のファンクションコールでのみチェックする
- 1　すべてのファンクションコールでチェックする
- 2　すべてのファンクションコールでチェックしない
- -1　設定状況の通知要求

ブレークチェックを行う特定のファンクションコールとその関数は次のとおり。

- $FF01　_dos_getchar 関数
- $FF02　_dos_putchar 関数
- $FF03　_dos_cominp 関数
- $FF04　_dos_comout 関数
- $FF05　_dos_prnout 関数
- $FF08　_dos_getc 関数
- $FF09　_dos_print 関数
- $FF0A　_dos_gets 関数
- $FF0B　_dos_keysns 関数
- $FF0C　_dos_kflush 関数
- $FF1D　_dos_fputc 関数
- $FF1E　_dos_fput 関数

_dos_breakck 関数は DOS コール 0xFF33 を発行することによって処理される。

戻り値――ブレークチェックの設定状況を返す。戻り値は，*flag* に指定できる 0 ～ 2 の値のいずれかになる。

規　格――*XC*

関連項目――_dos_cominp，_dos_comout，_dos_fput，_dos_fputc，_dos_getc，_dos_getchar，_dos_gets，_dos_keysns，_dos_kflush，_dos_print，_dos_prnout，_dos_putchar

_dos_change_pr

用　　途――バックグラウンドプロセスの実行権を放棄する。

書　　式――#include <sys/dos.h>

```
void _dos_change_pr (void);
```

解　　説――_dos_change_pr関数はバックグラウンドプロセスの実行権を放棄する。その結果，現在のタスクは中断し，次のタスクへの切り替えが行われる。

_dos_change_pr関数はDOSコール0xFFFFを発行することによって処理される。

戻り値――なし。

規　　格――*XC*

関連項目――_dos_get_pr，_dos_kill_pr，_dos_open_pr，_dos_send_pr，_dos_sleep_pr，_dos_suspend_pr，_dos_time_pr

_dos_chdir

用　途——カレントディレクトリを変更する。

書　式——#include <sys/dos.h>

```
int _dos_chdir (const char *file);
```

解　説——_dos_chdir 関数は *file* で指定したディレクトリに，カレントディレクトリを変更する。

_dos_chdir 関数は DOS コール 0xFF3B を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

規　格——*XC*

関連項目——_dos_mkdir，_dos_rmdir

_dos_chgdrv

用　　途——カレントドライブを変更する。

書　　式——#include <sys/dos.h>

```
void _dos_chgdrv (int drive);
```

解　　説——_dos_chgdrv 関数は *drive* で指定したドライブにカレントドライブを変更する。*drive* にはドライブ A:ならば 0，ドライブ B:ならば 1，ドライブ C:ならば 2，以下この順番でドライブ Z:まで設定できる。

_dos_chgdrv 関数は DOS コール 0xFF0E を発行することによって処理される。

戻り値——*drive* として指定可能な最大ドライブ番号を返す。ただし，*drive* よりも小さな値が返った場合はエラーになる。

規　　格——*XC*

_dos_chmod

用　　途——ファイル属性を変更する。

書　　式——#include <sys/dos.h>

```
dosmode_t _dos_chmod (const char *file, int atr);
```

解　　説——_dos_chmod 関数は *file* で指定したファイルのファイル属性を *atr* に変更する。

atr に指定できる値は次のとおり。ただし，ビット 3 ～ 4 は互いに排他である。

- ビット 0　読み込み専用
- ビット 1　隠しファイル
- ビット 2　システムファイル
- ビット 3　ボリュームラベル
- ビット 4　ディレクトリ
- ビット 5　通常のファイル

_dos_chmod 関数は DOS コール 0xFF43 を発行することによって処理される。

戻 り 値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

規　　格——*XC*

_dos_cinsns

用　途——RS-232C からの入力が可能かどうかを調べる。

書　式——

```
#include <sys/dos.h>
int _dos_cinsns (void);
```

解　説——_dos_cinsns 関数は，現在 RS-232C からの入力が可能であるかどうかを調べる。

_dos_cinsns 関数は DOS コール 0xFF12 を発行することによって処理される。

戻り値——RS-232C からの入力が可能ならば 0 以外を返し，不可能ならば 0 を返す。

規　格——*XC*

_dos_close

用　途——ファイルをクローズする。

書　式——

```
#include <sys/dos.h>
int _dos_close (int fildes);
```

解　説——_dos_close 関数は *fildes* で指定したファイルをクローズする。

_dos_close 関数は DOS コール 0xFF3E を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

規　格——*XC*

_dos_cominp

用　　途——RS-232C から 1 文字入力する。

書　　式——#include <sys/dos.h>

```
int _dos_cominp (void);
```

解　　説——_dos_cominp 関数は RS-232C から 1 文字読み込む。このときブレークチェックが行われる。

_dos_cominp 関数は DOS コール 0xFF03 を発行することによって処理される。

戻 り 値——RS-232C から読み込まれた文字の文字コードを返す。

注　　意——ブレークチェックで検査される文字は CTRL+C (ブレーク) のみである。

規　　格——*XC*

関連項目——_dos_cinsns, _dos_comout

_dos_common

用　途——Human68k の common 領域を操作する。

書　式——

```
#include <sys/dos.h>
int _dos_common_ck (const char *name);
int _dos_common_del (const char *name);
int _dos_common_fre (const char *name, int pos,
                     int id_psp, int len);
int _dos_common_lk (const char *name, int pos,
                    int id_psp, int len);
int _dos_common_rd (const char *name, int pos,
                    char *buff, int len);
int _dos_common_wt (const char *name, int pos,
                    const char *buff, int len);
```

解　説——_dos_common_ck 関数は，**Human68k** の common 領域において *name* で指定したブロックが存在するかどうかを調べる。

_dos_common_del 関数は，**Human68k** の common 領域において *name* で指定したブロックを削除する。

_dos_common_fre 関数は，**Human68k** の common 領域において *name* で指定したブロック内の *pos* で指定したデータのロックを解除する。ロックを解除すると，*id_psp* で指定したプロセス以外のプロセスからもデータをアクセスできるようになる。*name* はブロック名，*pos* はブロック内のデータの位置，*id_psp* はロックしたときにつけたプロセス ID，*len* はロックしたときに指定したバイト数を意味する。_dos_common_fre 関数で指定する引数は，_dos_common_lk 関数で指定した引数と完全に同じでなくてはならない。これは，ロックしたプロセス以外がロック解除を行うのを防ぐためである。

_dos_common_lk 関数は，**Human68k** の common 領域において *name* で指定したブロック内のさらに *pos* で指定したデータをロックする。ロックすると，*id_psp* で指定したプロセス以外のプロセスからはデータをアクセスできなくなる。*name* はブロック名，*pos* はブロック内のデータの位置，*id_psp* はロックするプロセスのプロセス ID，*len* はロックするバイト数を意味する。*id_psp* には必ず自プロセスのプロセス ID を指定すること。もしも適当に *id_psp* を指定すると，自分自身でもロックを解除できなくなる可能性があるからである。また，ロック解除は自分自身以外のプロセスからは不可能なので，ロックした場合は必ずロックを解除すること。ロックを解除せずにプロセスを終了した場合，永久にロック解除ができなくなる。

_dos_common_rd 関数は，**Human68k** の common 領域において *name* で指定したブロックのさらに *pos* で指定したバイト位置から，*len* バイト分 *buff* で指定する領域へ読み込む。

_dos_common_wt 関数は，**Human68k** の common 領域において *name* で指定したブロックのさらに *pos* で指定したバイト位置から，*len* バイト分 *buff* で指定する領域のデータを書き出す。*len* に 0 を指定するとブロックを切り詰める。

これらの関数は DOS コール 0xFF55(0xFF85) を発行することによって処理される。

戻り値——_dos_common_ck 関数は，*name* で指定したブロックが存在する場合はそのブロックの大きさを返し，存在しない場合は負の値を返す。

_dos_common_rd 関数は，成功した場合は読み出したバイト数を返し，失敗した場合は負の値を返す。

_dos_common_wt 関数は，成功した場合は書き出したバイト数を返し，失敗した場合は負の値を返す。

これ以外の関数では，失敗した場合は負の値を返す。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

_dos_comout

用　　途――RS-232C へ 1 文字出力する。

書　　式――#include <sys/dos.h>

```
void _dos_comout (int code);
```

解　　説――_dos_comout 関数は *code* で指定した 1 文字を RS-232C へ出力する。このときブレークチェックが行われる。

_dos_comout 関数は DOS コール 0xFF04 を発行することによって処理される。

戻 り 値――なし。

注　　意――ブレークチェックで検査される文字は CTRL+C (ブレーク) のみである。

規　　格――*XC*

関連項目――_dos_cominp, _dos_coutsns

_dos_conctrl

用　途――CONデバイスの出力を直接制御する。

書　式――

```
#include <sys/dos.h>
void _dos_c_putc (int code);
int _dos_c_print (const char *message);
int _dos_c_color (int atr);
int _dos_c_locate (int x, int y);
int _dos_c_down_s (void);
int _dos_c_up_s (void);
int _dos_c_down (int cnt);
int _dos_c_up (int cnt);
int _dos_c_right (int cnt);
int _dos_c_left (int cnt);
int _dos_c_cls_ed (void);
int _dos_c_cls_st (void);
int _dos_c_cls_al (void);
int _dos_c_era_ed (void);
int _dos_c_era_st (void);
int _dos_c_era_al (void);
int _dos_c_ins (int cnt);
int _dos_c_del (int cnt);
int _dos_c_fnkmod (int mode);
int _dos_c_window (int ys, int yl);
int _dos_c_width (int mode);
int _dos_c_curon (void);
int _dos_c_curoff (void);
```

解　説――_dos_c_putc関数は*code*で指定した1バイトのデータを表示する。

_dos_c_print関数は*message*で指定した文字列を表示する。

_dos_c_color関数は*atr*で指定した属性を設定する。設定可能なカラー属性は次のとおり。

- 0　黒
- 1　水色
- 2　黄色
- 3　白
- 4　黒

- 5　水色の強調
- 6　黄色の強調
- 7　白の強調
- 8　黒
- 9　水色のリバース
- 10　黄色のリバース
- 11　白のリバース
- 12　黒
- 13　水色の強調，リバース
- 14　黄色の強調，リバース
- 15　白の強調，リバース

_dos_c_locate 関数は *x* および *y* で指定した位置へカーソルを設定する。

_dos_c_down_s 関数はカーソル位置を１行下へ移動する。このとき，最終行より下に移動する場合はスクロールアップする。

_dos_c_up_s 関数はカーソル位置を１行上へ移動する。このとき，先頭行より上に移動する場合はスクロールダウンする。

_dos_c_down 関数はカーソル位置を *cnt* 行下へ移動する。このとき，最終行より下に移動しようとした場合でもスクロールアップは行われない。また，*cnt* に０を指定した場合は１を指定したと解釈される。

_dos_c_up 関数はカーソル位置を *cnt* 行上へ移動する。このとき，先頭行より上に移動しようとした場合でもスクロールダウンは行われない。また，*cnt* に０を指定した場合は１を指定したと解釈される。

_dos_c_right 関数はカーソル位置を *cnt* 桁右へ移動する。このとき，再右端行より右に移動しようとした場合でも左スクロールは行われない。また，*cnt* に０を指定した場合は１を指定したと解釈される。

_dos_c_left 関数はカーソル位置を *cnt* 桁左へ移動する。このとき，再左端行より左に移動しようとした場合でも右スクロールは行われない。また，*cnt* に０を指定した場合は１を指定したと解釈される。

_dos_c_cls_ed 関数は，現在のカーソル位置から最終行右端までのテキスト画面をクリアする。

_dos_c_cls_st 関数は，先頭行左端から現在のカーソル位置までのテキスト画面をクリアする。

_dos_c_cls_al 関数はテキスト画面全体をクリアする。クリア後カーソル位置はホームポジションへ移動する。

_dos_c_era_ed 関数は，現在のカーソル位置からカーソルがある行の右端までをクリアする。

_dos_c_era_st 関数は，現在カーソルのある行の左端からカーソル位置までをクリアする。

_dos_c_era_al 関数は，現在カーソルのある行をクリアする。

_dos_c_ins 関数は，現在カーソルのある行の下に *cnt* で指定した行数の空白行を挿入する。また，*cnt* に 0 を指定した場合は 1 を指定したと解釈される。

_dos_c_del 関数は，現在カーソルのある行から *cnt* で指定した行数削除する。また，*cnt* に 0 を指定した場合は 1 を指定したと解釈される。

_dos_c_fnkmod 関数は，ファンクションキー行を *mode* で指定したモードに設定する。ただし，この指定を行うとスクロール範囲の設定がクリアされる。*mode* には次の値を設定することができる。

- 0　ファンクションキー表示 (スクロール範囲は 0 から 31 行)
- 1　シフトファンクションキー表示 (スクロール範囲は 0 から 31 行)
- 2　何も表示しない (スクロール範囲は 0 から 31 行)
- 3　通常の行とする (スクロール範囲は 0 から 32 行)
- -1　現在の設定モードを返す

_dos_c_window 関数はスクロール範囲を設定する。このときカーソルはホームポジションへ移動する。また，*ys* と *yl* を足した値はファンクションキー行のモードが 3 のときに 32 まで，それ以外のときは 31 までとなるように設定しなくてはならない。スクロール範囲の設定を行うと，絶対座標(0, ys) が論理座標(0, 0) となる。

_dos_c_width 関数は画面モードの設定を行う。*mode* には次の値を設定することができる。

- 0　高解像度 768 × 512　グラフィックなし
- 1　高解像度 768 × 512　グラフィック 16 色
- 2　高解像度 512 × 512　グラフィックなし
- 3　高解像度 512 × 512　グラフィック 16 色
- 4　高解像度 512 × 512　グラフィック 256 色
- 5　高解像度 512 × 512　グラフィック 65536 色
- -1　現在のモードを返す

_dos_c_curon 関数はカーソルを表示状態にする。

_dos_c_curoff 関数はカーソルを非表示状態にする。

なお，これらの機能は CON デバイスによってサポートされている機能である。

これらの関数は DOS コール 0xFF23 を発行することによって処理される。

戻り値——_dos_c_width 関数は設定変更前の画面モードを返す。

それ以外の関数は，正常終了した場合は 0 を返す。

規　格——*XC*

_dos_consns

用　途——画面への出力が可能かどうかを調べる。

書　式——#include <sys/dos.h>

```
int _dos_consns (void);
```

解　説——_dos_consns 関数は，現在画面への出力が可能であるかどうかを調べる。

_dos_consns 関数は DOS コール 0xFF10 を発行することによって処理される。

戻り値——画面への出力が可能ならば 0 以外を返し，不可能ならば 0 を返す。

規　格——*XC*

_dos_coutsns

用　　途——RS-232Cへの出力が可能かどうかを調べる。

書　　式——#include <sys/dos.h>

```
int _dos_coutsns (void);
```

解　　説——_dos_coutsns関数は，現在RS-232Cへの出力が可能であるかどうかを調べる。

_dos_coutsns関数はDOSコール0xFF13を発行することによって処理される。

戻り値——RS-232Cへの出力が可能ならば0以外を返し，不可能ならば0を返す。

規　　格——*XC*

_dos_create

用　途――ファイルを新規に作成する。

書　式――
```
#include <sys/dos.h>
int _dos_create (const char *file, dosmode_t atr);
```

解　説――_dos_create 関数は *file* で指定したファイルを *atr* のファイル属性で新規に作成する。すでに存在している場合でもエラーとならず，強制的に作成する。

atr に指定できる値は次のとおり。ただし，ビット3～4は互いに排他である。

- ビット0　読み込み専用
- ビット1　隠しファイル
- ビット2　システムファイル
- ビット3　ボリュームラベル
- ビット4　ディレクトリ
- ビット5　通常のファイル

_dos_create 関数は DOS コール 0xFF3C を発行することによって処理される。

戻り値――成功した場合は作成したファイルハンドルを返し，失敗した場合は負の値を返す。

規　格――*XC*

_dos_ctlabort

用　途——CTRL+C アボート処理へジャンプする。

書　式——#include <sys/dos.h>

```
void __volatile _dos_ctlabort (void);
```

解　説——_dos_ctlabort 関数は，CTRL+C (ブレーク) が押された場合の処理へ実行を移すことができる。

_dos_ctlabort 関数は DOS コール 0xFFF1 を発行することによって処理される。

戻り値——なし。

規　格——*Project LIBC Group*

_dos_curdir

用　途――カレントディレクトリを取得する。

書　式――
```
#include <sys/dos.h>
int _dos_curdir (int drive, const char *buff);
```

解　説――_dos_curdir 関数は，*drive*で指定したドライブのカレントディレクトリを *buff* へセットする。*drive*にはカレントドライブならば0，ドライブA:ならば1，ドライブB:ならば2，以下この順番でドライブZ:まで設定できる。

*buff*にはドライブ名は含まれず，また先頭のパス区切り記号は省かれる。たとえば，カレントディレクトリが“/DIR/SUBDIR”だった場合，*buff*には“DIR/SUBDIR”と設定される。

_dos_curdir 関数はDOSコール0xFF47を発行することによって処理される。

戻り値――成功した場合は0を含む正の値を返し，失敗した場合は負の値を返す。

規　格――*XC*

_dos_curdrv

用　　途――カレントドライブの番号を調べる。

書　　式――

```
#include <sys/dos.h>
int _dos_curdrv (void);
```

解　　説――_dos_curdrv 関数はカレントドライブのドライブ番号を返す。

_dos_curdrv 関数は DOS コール 0xFF19 を発行することによって処理される。

戻 り 値――カレントドライブのドライブ番号を返す。ドライブ番号は A: ドライブを 0 とした番号である。

規　　格――*XC*

_dos_delete

用　途——ファイルを削除する。

書　式——
```
#include <sys/dos.h>
int _dos_delete (const char *file);
```

解　説——_dos_delete 関数は *file* で指定したファイルを削除する。

file にはワイルドカードやディレクトリは指定できない。また，オープンされているファイルを削除しようとした場合はエラーとなる。

_dos_delete 関数は DOS コール 0xFF41 を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

規　格——*XC*

_dos_diskred

用　途——ブロックデバイスへの直接入力を行う。

書　式——#include <sys/dos.h>

```
void _dos_diskred (void *addr,
                   int drive, int sector, int seclen);
void _dos_diskred2 (void *addr,
                    int drive, int sector, int seclen);
```

解　説——_dos_diskred関数はブロックデバイスの直接入力を行う。

読み込みはセクタ単位で行われるので，*addr*からのバッファのサイズは，読み込みを行うデバイスの1セクタのサイズの整数倍となるように設定する。なお標準のデバイスドライバでは，1セクタは1024バイトなので1024の倍数を指定する。*drive*には，カレントドライブならば0，ドライブA:ならば1，ドライブB:ならば2，以下この順番でドライブZ:まで設定できる。

仮想ドライブや仮想ディレクトリに割り当てた実ドライブにはアクセスできない。

*sector*および*seclen*は65536までの数値で指定すること。これ以上のセクタ数/セクタ長を指定することはできない。これ以上の大容量ドライブに対するアクセスを行う場合は，_dos_diskred2関数を使用すること。

_dos_diskred2関数は大容量ブロックデバイスに対しての直接入力を行う。

_dos_diskred2関数に対しては，*sector*および*seclen*に対して65536を越えたロングワードの値を指定することができる。これ以外の引数に関しては，_dos_diskred関数と同じである。

これらの関数はDOSコール0xFFF3を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_dos_diskwrt，_dos_diskwrt2

_dos_diskwrt

用　　途——ブロックデバイスへの直接出力を行う。

書　　式——
```
#include <sys/dos.h>
void _dos_diskwrt (void *addr,
                   int drive, int sector, int seclen);
void _dos_diskwrt2 (void *addr,
                    int drive, int sector, int seclen);
```

解　　説——_dos_diskwrt 関数はブロックデバイスの直接出力を行う。

書き込みはセクタ単位で行なわれるので，*addr* からのバッファのサイズは，書き込みを行うデバイスの1セクタのサイズの整数倍となるように設定する。なお標準のデバイスドライバでは1セクタは1024バイトなので1024の倍数を指定する。*drive* には，カレントドライブならば0，ドライブ A:ならば1，ドライブ B:ならば2，以下この順番でドライブ Z:まで設定できる。

仮想ドライブや仮想ディレクトリに割り当てた実ドライブにはアクセスできない。

sector および *seclen* には65536までの数値で指定すること。これ以上のセクタ数/セクタ長を指定することはできない。これ以上の大容量ドライブに対するアクセスを行う場合は_dos_diskwrt2 関数を使用すること。

_dos_diskwrt2 関数は大容量ブロックデバイスに対しての直接出力を行う。

_dos_diskwrt2 関数に対しては *sector* および *seclen* に対して65536を越えたロングワードの値を指定することができる。これ以外の引数に関しては_dos_diskwrt 関数と同じである。

これらの関数は DOS コール 0xFFF4 を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

関連項目——_dos_diskred，_dos_diskred2

_dos_drvctrl

用　途——ドライブ状態のチェックおよび設定を行う。

書　式——

```
#include <sys/dos.h>
void _dos_drvctrl (int mode, int drive);
```

解　説——_dos_drvctrl関数は，*drive*で指定したドライブの状態センスまたは設定を行う。*drive*にはカレントドライブならば0，ドライブA:ならば1，ドライブB:ならば2，以下この順番でドライブZ:まで設定できる。*mode*の設定値は次のとおり。

- 0　ドライブ状態のセンス
- 1　イジェクトする。ただし該当ドライブにオープンファイルが存在する場合はイジェクトしない
- 2　イジェクトを禁止する(mode=1の指定より優先)
- 3　イジェクトを許可する。ただし該当ドライブにオープンファイルが存在する場合は，クローズはされないが自動的にバッファフラッシュされる
- 4　ディスクがセットされていないときにLEDを点滅
- 5　ディスクがセットされていないときにLEDを消灯
- 6 ～ 8　使用不可
- 9　該当ドライブのカレントディレクトリをルートディレクトリとし，FAT検索も先頭からにする。ただし，オープンファイルが存在する場合はエラーとなる
- 10　該当ドライブのFAT検索を先頭からにする

_dos_drvctrl関数はDOSコール0xFF0Fを発行することによって処理される。

戻り値——*drive*で指定したドライブの状態をビットフィールドで返す。各ビットの内容は次のとおり。

- ビット0　メディア誤挿入
- ビット1　メディア挿入
- ビット2　ノットレディ
- ビット3　ライトプロテクト
- ビット4　ユーザによるイジェクト禁止
- ビット5　バッファあり
- ビット6　イジェクト禁止
- ビット7　LED点滅

いずれも該当ビットが1のとき，上記の状態となる。

またビット2，3については，*mode*に0(ドライブ状態のセンス)を指定し，かつメディアが挿入されている場合にのみ有効となる。

規　　格――*XC*

_dos_drvxchg

用　　途——ドライブを入れ替える。

書　　式——#include <sys/dos.h>

```
void _dos_drvxchg (int old, int new);
```

解　　説——_dos_drvxchg 関数は *old* と *new* で指定したドライブを入れ替える。*old* と *new* には，カレントドライブならば 0，ドライブ A: ならば 1，ドライブ B: ならば 2，以下この順番でドライブ Z: まで設定できる。

_dos_drvxchg 関数は DOS コール 0xFF34 を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_dos_dskfre

用　　途――ディスクの残り容量を調べる。

書　　式――#include <sys/dos.h>

```
int _dos_dskfre (int drive, struct _freeinf *buff);
```

解　　説――_dos_dskfre 関数は *drive* で指定したドライブの残り容量を調べ，その結果を *buff* で指定した _freeinf 構造体へセットする。*buff* に指定する _freeinf 構造体は次のとおり。

```
struct _freeinf {
    unsigned short free; /* 使用可能なクラスタ数 */
    unsigned short max;  /* 総クラスタ数 */
    unsigned short sec;  /* 1 クラスタ当たりのセクタ数 */
    unsigned short byte; /* 1 セクタ当たりのバイト数 */
};
```

_dos_dskfre 関数は DOS コール 0xFF36 を発行することによって処理される。

戻 り 値――成功した場合は使用可能なバイト数 (最大 2G バイト) を返し，失敗した場合は負の値を返す。

規　　格――*XC*

_dos_dup

用　　途——ファイルハンドルを複写する。

書　　式——#include <sys/dos.h>

```
int _dos_dup (int fildes);
```

解　　説——_dos_dup 関数は *fildes* で指定したファイルハンドルを複写して，新しいファイルハンドルを作成する。

複写されたファイルハンドルは複写元のファイルハンドルと同一のファイルに対してアクセスするので，どちらのファイルハンドルを使用しても，同じファイルに対してファイル操作が可能となる。

_dos_dup 関数は標準入力/標準出力を切り替える(リダイレクトなど)ときに，切り替え前のファイルハンドルを保存する場合などに使用する。

_dos_dup 関数は DOS コール 0xFF45 を発行することによって処理される。

戻 り 値——成功した場合は複写された新しいファイルハンドルを返し，失敗した場合は負の値を返す。

規　　格——*XC*

関連項目——_dos_dup0，_dos_dup2

_dos_dup0

用　途——ファイルハンドルを強制的に複写する。

書　式——#include <sys/dos.h>

```
int _dos_dup0 (int fildes, int newdes);
```

解　説——_dos_dup0 関数は *fildes* のファイルハンドルを，*newdes* のファイルハンドルへ強制複写する。

_dos_dup0 関数は，**Human68k** がデフォルトでオープンしている 0 ～ 4 までのファイルハンドルの切り替えを行う (CTTY) ためのものである。標準入出力/標準エラー出力を _dos_dup 関数や _dos_dup2 関数によって切り替えた (リダイレクトなど) 後で _dos_close 関数を実行させると，_dos_dup0 関数で切り替えたファイルハンドルへ戻される。

_dos_dup0 関数は DOS コール 0xFF2F を発行することによって処理される。

戻り値——成功した場合は *newdes* の前の値を返し，失敗した場合は負の値を返す。

規　格——*XC*

関連項目——_dos_close，_dos_dup，_dos_dup2

_dos_dup2

用　途——ファイルハンドルを複写する。

書　式——
```
#include <sys/dos.h>
int _dos_dup2 (int fildes, int newdes);
```

解　説——_dos_dup2 関数は *fildes* で指定したファイルハンドルを，*newdes* のファイルハンドルに強制複写する。ただし，*newdes* で指定したファイルハンドルがオープンされている場合は，クローズしてから複写する。

複写されたファイルハンドルは複写元のファイルハンドルと同一のファイルに対してアクセスするので，どちらのファイルハンドルを使用しても，同じファイルに対してファイル操作が可能となる。

_dos_dup2 関数は標準入力/標準出力を切り替える (リダイレクトなど) ときに，切り替え前のファイルハンドルを保存する場合などに使用する。

_dos_dup2 関数は DOS コール 0xFF46 を発行することによって処理される。

戻り値——失敗した場合は負の値を返す。

規　格——*XC*

関連項目——_dos_dup，_dos_dup0

_dos_errabort

用　途――エラーアボート処理へジャンプする。

書　式――#include <sys/dos.h>

```
void __volatile _dos_errabort (void);
```

解　説――_dos_errabort 関数は，エラーによってプロセスをアボートする場合の処理へ実行を移すことができる。

_dos_errabort 関数は DOS コール 0xFFF2 を発行することによって処理される。

戻り値――なし。

規　格――***Project LIBC Group***

E

_dos_exec

用　　途——プログラムをロード/実行する。

書　　式——

```
#include <sys/dos.h>
int _dos_loadexec (const char *file,
                   const struct _comline *comline,
                   const char *envptr);
int _dos_load (const char *file, const struct _comline *comline,
               const char *envptr);
int _dos_pathchk (const char *file,
                  const struct _comline *comline,
                  const char *envptr);
int _dos_loadonly (const char *file, const void *loadaddr,
                   const void *limitaddr);
int _dos_execonly (void *jmpaddr);
int _dos_bindno (const char *file1, const char *file2);
int _dos_exec2 (int mode, const char *file, const char *p1,
                const char *p2);
```

解　　説——_dos_loadexec 関数は *comline* で指定したコマンドラインと *envptr* で指定した環境に基づいて，*file* で指定したプログラムをロード/実行する。

envptr に 0 を指定すると，呼び出しプロセスの環境がそのまま引き継がれる。

file の最上位 8 ビットの意味は次のとおり。

- 0　拡張子 .R/.Z/.X にしたがいロードする
- 1　R 形式のファイルとしてロードする
- 2　Z 形式のファイルとしてロードする
- 3　X 形式のファイルとしてロードする

comline に指定する _comline 構造体は次のとおり。

```
struct _comline {
  unsigned char len;   /* コマンドラインの長さ */
  char buffer[255];    /* コマンドライン文字列 */
};
```

_dos_load 関数は *comline* で指定したコマンドラインと *envptr* で指定した環境に基づいて，*file* で指定したプログラムをロードする。

envptr に 0 を指定すると，呼び出しプロセスの環境がそのまま引き継がれる。

*file*の最上位8ビットの意味は_dos_loadexec関数と同じである。

_dos_pathchk関数は*envptr*で指定した環境から環境変数pathをサーチし，その環境変数pathで指定したパスから*file*をサーチし，*file*のコマンド行をファイル名(ドライブ名，パス名つき)とコマンドラインに分けて，それぞれ*file*と*comline*が示す領域にセットする。

*file*には90バイト以上，*comline*には256バイト以上の領域が必要となる。*envptr*に0を指定すると，呼び出しプロセスの環境をサーチする。この関数を実行後，*comline*と*file*を指定して_dos_loadexec関数，または_dos_load関数を実行した場合，すでに通ったパスであれば，どのパスにおいても簡単にプログラムを実行することができる。

*envptr*に指定する環境へのポインタはグローバル変数environが指定するC独自の環境ではなく，**Human68k**の管理する環境へのポインタでなくてはならない。そのため*envptr*に0以外を指定する場合は注意が必要である。

_dos_loadonly関数は*loadaddr*から*limitaddr*までのメモリに，*file*で指定したプログラムをロードする。

*file*の最上位8ビットの意味は_dos_loadexec関数と同じである。

_dos_execonly関数は_dos_load関数でロードしたプログラムを実行する。*jmpaddr*には_dos_load関数の戻り値を指定する。

_dos_bindno関数は*file1*で指定したバインドファイルに含まれる*file2*で指定したモジュールのモジュール番号を取得する。

オーバーレイXファイルに含まれる個々のファイルを実行するためには，各ファイルに割りあてられたモジュール番号を指定しなくてはならないので，_dos_bindno関数でモジュール番号を求める。

_dos_exec2関数はプログラムのロード，実行，モジュール番号の取得を行う。

*mode*に指定する値は次のとおり。

- ビット15 ～ 8　モジュール番号
- ビット7 ～ 0　モード番号

*mode*に指定できる値とその内容および*p1*，*p2*との関係を次に示す。

- 0　*p1*で指定したコマンドラインと*p2*で指定した環境ポインタを用いて，*file*で指定したファイルをロード，実行する。*p2*の環境ポインタを0とすると，現在の環境を引き継ぐ
- 1　*p1*で指定したコマンドラインと，*p2*で指定した環境ポインタを用いて，*file*で指定したファイルをロードする。*p2*の環境ポインタを0とすると，現在の環境を引き継ぐ
- 2　*p2*で指定した環境より環境変数pathを検索し，それを基に*file*で指定したコマンド行(コマンド名+オプションなど)をコマンド名のパスとコ

マンドラインに分離し，それぞれ *file* と *p1* の指定する領域へセットする。*p2* の環境ポインタを 0 とすると，現在の環境を引き継ぐ。

mode を 2 として _dos_exec2 関数を実行した後に，*mode* を 0 または 1 として _dos_exec2 関数を実行すれば，環境変数 path に設定されているディレクトリにあるコマンドを簡単に実行できる。

- 3　*p1* にロードアドレス，*p2* にリミットアドレスを指定して，*file* で指定したプログラムをロードする
- 4　*file* に実行アドレスを指定し，すでにロードされているプログラムを実行する。この場合，*mode* のモジュール番号は 0 とする
- 5　*file* で指定したオーバーレイファイルより，*p1* で指定したモジュールを検索し，見つかった場合にそのモジュール番号を返す。この場合，*mode* のモジュール番号は 0 とする

mode が 0，1，2，3 のとき，*file* の最上位 8 ビットの意味は _dos_loadexec 関数と同じである。

これらの関数は DOS コール 0xFF4B を発行することによって処理される。

戻 り 値――_dos_loadexec 関数は，成功した場合は実行したプロセスの終了コードを返し，失敗した場合は負の値を返す。

_dos_load 関数は，成功した場合はロードしたプログラムのアドレスを返し，失敗した場合は負の値を返す。

_dos_pathchk 関数は，失敗した場合は負の値を返す。

_dos_loadonly 関数は，成功した場合はロードしたプログラムのサイズを返し，失敗した場合は負の値を返す。

_dos_execonly 関数は，成功した場合は実行したプロセスの終了コードを返し，失敗した場合は負の値を返す。

_dos_bindno 関数は，成功した場合は *file2* のモジュール番号を戻り値のビット 15 ～ 8 に返し，失敗した場合は負の値を返す。

_dos_exec2 関数は成功した場合，*mode* が 0，4 ならば子プロセスの終了コードを，1 ならばロードしたプログラムの実行アドレスを返す。戻り値が負の値ならばエラーが起こったことを示し，ロードや実行は行われない。

また *mode* が 2，3 ならば失敗した場合に負の値を返し，5 ならばモジュール番号を戻り値のビット 15 ～ 8 に返す。いずれも失敗した場合は負の値を返す。

規　　格――*XC*

_dos_exit

用　途——現在のプロセスを終了し，親プロセスへ復帰する。

書　式——#include <sys/dos.h>

```
void __volatile _dos_exit (void);
```

解　説——_dos_exit 関数は現在のプロセスを終了し，呼び出した親プロセスへ復帰する。このとき，オープンされているファイル (子プロセスがオープンしたファイルも含む) をすべてクローズする。

_dos_exit 関数は DOS コール 0xFF00 を発行することによって処理される。

戻り値——親プロセスへは 0 が返される。

注　意——親プロセスへはいつでも 0 が返される。戻り値の指定が必要な場合は，_dos_exit2 関数を使用すること。

規　格——*XC*

関連項目——_dos_exit2，_dos_keeppr

_dos_exit2

用　途——現在のプロセスを終了し，親プロセスへ復帰する。

書　式——#include <sys/dos.h>

```
void __volatile _dos_exit2 (int code);
```

解　説——_dos_exit2 関数は現在のプロセスを終了し，呼び出した親プロセスへ復帰する。ただし，プロセスの終了時には終了コードとして *code* が親プロセスへ返される。このとき，オープンされているファイル(子プロセスがオープンしたファイルも含む) をすべてクローズする。

_dos_exit2 関数は DOS コール 0xFF4C を発行することによって処理される。

戻り値——呼び出したプロセスへは戻らない。親プロセスへは *code* が返される。

規　格——*XC*

関連項目——_dos_exit，_dos_keeppr

_dos_fatchk

用　途――指定ファイルの使用セクタが連続しているかどうかを調べる。

書　式――#include <sys/dos.h>

```
int _dos_fatchk (const char *file, unsigned short *buff);
```

解　説――_dos_fatchk 関数は *file* で指定したファイルについて，その使用している FAT の使用状況を *buff* へ格納し，*buff* の使用バイト数を返す。

つまり戻り値が 8 の場合に，*file* で指定したファイルは連続した FAT を使用していることになる。また，使用セクタ番号を調べることで，ファイルをオープンしなくても _dos_diskred 関数や _dos_diskred2 関数を使用して，直接データを取得することが可能である。

_dos_fatchk 関数は DOS コール 0xFF17 を発行することによって処理される。

戻り値――成功した場合は *buff* の使用バイト数 (バッファの終端を示す 0 を含む) を返し，失敗した場合は負の値を返す。なお，*buff* は unsigned short 型の配列として使用される。設定されるデータは次のとおり。

- +0　ドライブ番号
- +2　先頭ブロックの先頭セクタ番号
- +4　ブロックが連続して使用しているセクタ数
- +6　次ブロックの先頭セクタ番号
- +8　ブロックが連続して使用しているセクタ数

　　⋮

- +n　終端を示す 0 (unsigned short 型)

ただし，ブロックの先頭セクタ番号が 0 の場合にバッファの終端とする。また，ドライブ番号は A:ドライブを 1 とした番号である。

注　意――大容量メディア上のファイルに対しては，_dos_fatchk2 関数を使用すること。_dos_fatchk2 関数は，セクタ番号およびセクタ数を unsigned long 型で返す。

規　格――*XC*

関連項目――_dos_diskred，_dos_diskred2，_dos_fatchk2

_dos_fatchk2

F

用　途――指定ファイルの使用セクタが連続しているかどうかを調べる。

書　式――

```
#include <sys/dos.h>
int _dos_fatchk2 (const char *file,
                  unsigned short *buff, int len);
```

解　説――_dos_fatchk2 関数は *file* で指定したファイルについて，その使用している FAT の使用状況を *buff* へ格納し，*buff* の使用バイト数を返す。

つまり戻り値が 14 の場合に，*file* で指定したファイルは連続した FAT を使用していることになる。

なお，*buff* に設定可能な最大バイト数は *len* で指定する。また，使用セクタ番号を調べることで，ファイルをオープンしなくとも _dos_diskred 関数や _dos_diskred2 関数を使用して，直接データを取得することが可能である。

_dos_fatchk2 関数は DOS コール 0xFF17 を発行することによって処理される。

戻り値――成功した場合は *buff* の使用バイト数 (バッファの終端を示す 0 を含む) を返し，失敗した場合は負の値を返す。なお，*buff* は先頭のドライブ番号が unsigned short 型，それ以降は unsigned long 型の配列として使用される。設定されるデータは次のとおり。

- +0　ドライブ番号
- +2　先頭ブロックの先頭セクタ番号
- +6　ブロックが連続して使用しているセクタ数
- +10　次ブロックの先頭セクタ番号
- +14　ブロックが連続して使用しているセクタ数

　　⋮

- +n　終端を示す 0 (unsigned short 型)

ただし，ブロックの先頭セクタ番号が 0 の場合にバッファの終端とする。また，ドライブ番号は A: ドライブを 1 とした番号である。

注　意――_dos_fatchk2 関数は _dos_fatchk 関数に対して，大容量メディアや特殊メディア上のファイルに対応している。

規　格――*XC*

関連項目――_dos_diskred，_dos_diskred2，_dos_fatchk

_dos_fflush

用　　途——ディスクのリセットを行う。

書　　式——#include <sys/dos.h>

void _dos_fflush (void);

解　　説——_dos_fflush 関数はディスクのリセットを行う。

ディスクバッファの内容をすべてフラッシュするが，オープンされているファイルのクローズは行わない。

_dos_fflush 関数は DOS コール 0xFF0D を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_dos_fgetc

用　途——ファイルハンドルから1バイト入力する。

書　式——#include <sys/dos.h>

```
void _dos_fgetc (int fildes);
```

解　説——_dos_fgetc 関数は *fildes* で指定したファイルハンドルから入力があるまで待ち，入力された1バイトを返す。

_dos_fgetc 関数は DOS コール 0xFF1B を発行することによって処理される。

戻り値——入力コードを返す。

規　格——*XC*

F

_dos_fgets

用　　途――ファイルハンドルから文字列を入力する。

書　　式――#include <sys/dos.h>

```
int _dos_fgets (struct _inpptr *inpptr, int fildes);
```

解　　説――_dos_fgets 関数は *fildes* で指定したファイルハンドルから，入力データ中に CR/LF コード（改行コード）が現れるまで文字列を入力する。入力中はブレークチェックを行わず，VOID/NEWLINE が入力されても改行しない。

入力された文字列は，*inpptr* で指定した _inpptr 構造体のメンバ length および buffer へ格納される。また，格納される最大入力文字数を _inpptr 構造体のメンバ max によって指定する。ただし，max で指定した最大入力文字数に達した場合は，最大入力文字数までを格納する。*inpptr* に指定する _inpptr 構造体は次のとおり。

```
struct _inpptr {
    unsigned char max;     /* 最大入力文字数 */
    unsigned char length; /* 実際の入力文字数 */
    char buffer[256];      /* 入力バッファ */
};
```

_dos_fgets 関数は DOS コール 0xFF1C を発行することによって処理される。

戻 り 値――*inpptr* が指定する _inpptr 構造体の buffer へ入力文字列を格納する。また，_inpptr 構造体の length には実際に入力された文字数が格納される。

関数の戻り値は，最後の CR/LF コードを含まない入力文字数である。

規　　格――*XC*

関連項目――_dos_gets，_dos_getss

_dos_filedate

用　　途――ファイル日付/時刻の読み込みと設定を行う。

書　　式――#include <sys/dos.h>

```
int _dos_filedate (int fildes, int filedate);
```

解　　説――_dos_filedate 関数は *fildes* で指定したファイルの日付を読み込み，設定する。

filedate に 0 を指定した場合，日付/時刻の読み込みを行い，これ以外の値を指定した場合に日付/時刻の設定を行う。

filedate および戻り値のビットフィールドの内容は次のとおり。

YYYYYYYM MMMDDDDD hhhhhmmm mmmsssss

- sssss　　秒 (0 ～ 29) (実際の値は 2 を乗ずる)
- mmmmmm　　分 (0 ～ 59)
- hhhhh　　時 (0 ～ 23)
- DDDDD　　日 (1 ～ 31)
- MMMM　　月 (1 ～ 12)
- YYYYYYY　　年 (0 ～ 99) (1980 年からの相対年数)

_dos_filedate 関数は DOS コール 0xFF57(0xFF87) を発行することによって処理される。

戻 り 値――*filedate* が 0 の場合には *fildes* で指定したファイルの日付/時刻を返す。ただし，戻り値の上位 16 ビットが 0xFFFF ならばエラーが発生したことを示す。

注　　意――日付/時刻の設定を行うためには，*fildes* で指定したファイルが書き込み可能なモードでオープンされていなくてはならない。また日付/時刻の設定を行った後で，*fildes* で指定したファイルに対して書き込みを行うと，設定した日付/時間が無効になるので注意すること。したがって設定を行う場合は，必ずクローズ直前に行うようにする。

互 換 性――**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格――*XC*

_dos_files

用　途――ファイルを検索する。

書　式――

```
#include <sys/dos.h>
void _dos_files (struct _filbuf *buff,
                 const char *file, int atr);
```

解　説――_dos_files 関数は *file* で指定したファイル名と *atr* で指定したファイル属性のファイルを検索し，*buff* へその情報をセットする。

file にはワイルドカードが指定できる。ワイルドカードを指定したとき，最初に見つかったファイルが目的のファイルではない場合は，_dos_nfiles 関数を使用して次のファイルをサーチすることができる。

atr に指定できる値は次のとおり。

- 0x01　　読み込み専用
- 0x02　　隠しファイル
- 0x04　　ボリューム ID
- 0x08　　システムファイ
- 0x10　　ディレクトリ
- 0x20　　通常のファイル

この *atr* で指定した属性とファイル属性の論理積 (AND) が 0 でないファイルはサーチの対象となる。したがってすべての属性のファイルを対象とするには，*atr* に 0x3F を指定する。*buff* に指定する _filbuf 構造体は次のとおり。

```
struct _filbuf {
    unsigned char searchatr; /* 検索するファイル属性 */
    unsigned char driveno;   /* ドライブ番号 */
    unsigned long dirsec;    /* ディレクトリエントリのセクタ番号 */
    unsigned short dirlft;   /* ディレクトリエントリの残りのセクタ数 */
    unsigned short dirpos;   /* ディレクトリエントリ内の位置 */
    char filename[8];        /* ファイルのノード名 */
    char ext[3];             /* ファイルの拡張子名 */
    unsigned char atr;       /* ファイル属性 */
    unsigned short time;     /* ファイル時刻 */
    unsigned short date;     /* ファイル日時 */
    unsigned int filelen;    /* ファイル長 */
    char name[23];           /* ファイル名 */
};
```

_dos_files 関数は DOS コール 0xFF4E を発行することによって処理される。

戻 り 値——成功した場合は0を含む正の値を返し，失敗した場合は負の値を返す。通常は0が返るが，特殊デバイスドライバが対象の場合は0以外の正の値の場合もある。

注　　意——読み込み専用の通常ファイルを検索対象としようとして 0x21 を *atr* に指定しても，すべての通常ファイルとすべての読み込み専用ファイルが対象となるので注意すること。

_filbuf 構造体の searchatr から ext までのメンバは **Human68k** が使用するので，ユーザは変更してはならない。変更した場合，_dos_nfiles 関数が使用できなくなる。

_filbuf 構造体のメンバ dirpos は *file* にワイルドカードを指定したときにだけ正しく設定され，それ以外の場合は 0xFFFF が設定される。また，dirpos が 0xFFFF の場合は _dos_nfiles 関数による再検索は行えない。

規　　格——*XC*

関連項目——_dos_nfiles

F

_dos_fnckey

用　　途——再定義可能キーの読み込みおよび設定を行なう。

書　　式——#include <sys/dos.h>

```
void _dos_fnckeygt (int fno, char *buff);
void _dos_fnckeyst (int fno, const char *buff);
```

解　　説——_dos_fnckeygt 関数は *fno* で指定した再定義可能キーの設定を読み込み，結果を *buff* の指す領域に格納する。

_dos_fnckeyst 関数は *fno* で指定した再定義可能キーを *buff* で指定したデータで再設定する。

このとき，読み込みまたは設定するキーによって *buff* の使用容量が変化する。*fno* と *buff* の関係は次のとおり。

fno	必要サイズ	キーの種類
0	712	すべてのキー
1~10	32 (31+'\0')	F1~F10
11~20	32 (31+'\0')	SHIFT+F1~SHIFT+F10
21	6 (5+'\0')	ROLL UP
22	6 (5+'\0')	ROLL DOWN
23	6 (5+'\0')	INS
24	6 (5+'\0')	DEL
25	6 (5+'\0')	UP
26	6 (5+'\0')	LEFT
27	6 (5+'\0')	RIGHT
28	6 (5+'\0')	DOWN
29	6 (5+'\0')	CLR
30	6 (5+'\0')	HELP
31	6 (5+'\0')	HOME
32	6 (5+'\0')	UNDO

なお，本機能はCONデバイスによってサポートされている機能である。

これら2つの関数はDOSコール0xFF21を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_dos_fputc

用　途——ファイルハンドルへ文字を出力する。

書　式——#include <sys/dos.h>

```
void _dos_fputc (int code, int fildes);
```

解　説——_dos_fputc 関数は *fildes* で指定したファイルハンドルへ *code* の文字を出力する。

出力デバイスがキャラクタデバイスの場合は，CTRL+C，CTRL+S，CTRL+P，CTRL+N についてブレークチェックが行われる。

_dos_fputc 関数は DOS コール 0xFF1D を発行することによって処理される。

戻り値——なし。

規　格——*XC*

F

_dos_fputs

用　　途——ファイルハンドルへ文字列を出力する。

書　　式——#include <sys/dos.h>

```
void _dos_fputs (const char *message, int fildes);
```

解　　説——_dos_fputs 関数は *fildes* で指定したファイルハンドルへ，*message* で指定した文字列を出力する。文字列は null 文字で終了していなければならない。

出力デバイスがキャラクタデバイスの場合は，CTRL+C，CTRL+S，CTRL+P，CTRL+N についてブレークチェックが行われる。

_dos_fputc 関数は DOS コール 0xFF1E を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_dos_get_pr

用　途——スレッドの管理情報を取得する。

書　式——#include <sys/dos.h>

```
int _dos_get_pr (int id, struct _prcptr *buff);
```

解　説——_dos_get_pr 関数は *id* で指定したスレッドの管理情報を取得する。

ただし，*id* に−1 を指定し，*buff* のメンバ name にバックグラウンドプロセスの名前を指定して_dos_get_pr 関数を実行した場合は，戻り値としてバックグラウンドプロセスの ID が，*buff* で指定した_prcctrl 構造体に管理情報が返される。また，*id* に−2 を指定すると，自分自身の ID が戻り値として返され，*buff* で指定した_prcctrl 構造体に管理情報が返される。

buff に指定する_prcctrl 構造体は次のとおり。

```
struct _prcptr {
    struct _prcptr *next_ptr;     /* 次の管理領域へのポインタ */
    unsigned char wait_flg;       /* normal=0x00/wait=0xFF */
    unsigned char counter;        /* 割り込みごとに減算される */
    unsigned char max_counter;    /* counter のプリセット値 */
    unsigned char doscmd;         /* DOS コール番号 */
    unsigned int psp_id;          /* プロセス ID */
    unsigned int usp_reg;         /* usp レジスタ */
    unsigned int d_reg[8];        /* データレジスタ */
    unsigned int a_reg[7];        /* アドレスレジスタ */
    unsigned short sr_reg;        /* sr レジスタ */
    unsigned int pc_reg;          /* pc レジスタ */
    unsigned int ssp_reg;         /* ssp レジスタ */
    unsigned short indosf;        /* システム予約 */
    unsigned int indosp;          /* システム予約 */
    struct _prcctrl *buf_ptr;     /* タスク間通信バッファへのポインタ */
    unsigned char name[16];       /* スレッドの名前 */
    long wait_time;               /* 待時間の残り (ミリ秒) */
};
```

_dos_get_pr 関数は DOS コール 0xFFFA を発行することによって処理される。

戻り値——成功した場合は求めるバックグラウンドプロセスの *id* を返し，失敗した場合は負の値を返す。

id が不正の場合は 0xFFFF00??が返るが，この 0x??の部分が指定できる *id* の最大値を示している。また，指定した名前のバックグラウンドプロセスが存在しない場合は−1 を返す。

規　格 —— *XC*

関連項目 —— _dos_change_pr, _dos_kill_pr, _dos_open_pr, _dos_send_pr, _dos_sleep_pr, _dos_suspend_pr, _dos_time_pr

_dos_getc

用　途——キーボードから１文字入力する。

書　式——#include <sys/dos.h>

```
int _dos_getc (void);
```

解　説——_dos_getc 関数はキーが押されるまで待ち，押されたキーコードを返す。このとき，ブレークチェックを行う。

_dos_getc 関数は DOS コール 0xFF08 を発行することによって処理される。

戻り値——入力されたキーコードを返す。

注　意——ブレークチェックで検査される文字には，CTRL+C (ブレーク)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー中止) がある。

規　格——*XC*

関連項目——_dos_getchar，_dos_inkey

_dos_getchar

用　途——標準入力から１文字読み込む。

書　式——#include <sys/dos.h>

```
int _dos_getchar (void);
```

解　説——_dos_getchar 関数は標準入力から１文字読み込み，読み込んだ文字を標準出力へエコーバックする。このときブレークチェックが行われる。

_dos_getchar 関数は DOS コール 0xFF01 を発行することによって処理される。

戻り値——標準入力から読み込まれた文字の文字コードを返す。

注　意——ブレークチェックで検査される文字には，CTRL+C (ブレーク)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー中止) がある。

規　格——*XC*

関連項目——_dos_getc，_dos_inkey，_dos_inpout，_dos_putchar

_dos_getdate

用　　途——現在の日付を取得する。

書　　式——#include <sys/dos.h>

```
int _dos_getdate (void);
```

解　　説——_dos_getdate 関数は現在の日付を返す。戻り値のビットフィールドの内容は次の通り。

```
00000000 00000www yyyyyyym mmmddddd
```

- ddddd　日 (1 ~ 31)
- mmmm　月 (1 ~ 12)
- yyyyyyy　年 (0 ~ 99) (1980 年からの相対年数)
- www　曜日 (0=日・・・6=土)

_dos_getdate 関数は DOS コール 0xFF2A を発行することによって処理される。

戻 り 値——現在の日付を返す。

規　　格——*XC*

関連項目——_dos_gettim2, _dos_gettime, _dos_setdate, _dos_settim2, _dos_settime

G

_dos_getdpb

用　途――ドライブパラメータブロックを取得する。

書　式――#include <sys/dos.h>

```
int _dos_getdpb (int drive, struct _dpbptr *dpbptr);
```

解　説――_dos_getdpb関数は，*drive*で指定したドライブの**ドライブパラメータブロック**(DPB)を*dpbptr*へセットする。*drive*にはカレントドライブならば0，ドライブA:ならば1，ドライブB:ならば2，以下この順番でドライブZ:まで設定できる。*dpbptr*に指定する_dpbptr構造体は次のとおり。

```
struct _dpbptr {
    unsigned char drive;        /* ドライブ番号 */
    unsigned char unit;         /* デバイスドライバで使うユニット番号 */
    unsigned short byte;        /* 1セクタ当たりのバイト数 */
    unsigned char sec;          /* 1クラスタ当たりのセクタ数$-$1 */
    unsigned char shift;        /* クラスタ，セクタ間のシフト数 */
    unsigned short fatsec;      /* FATの先頭セクタ番号 */
    unsigned char fatcount;     /* FAT領域の個数 */
    unsigned char fatlen;       /* FATの占めるセクタ数 */
    unsigned short dircount;    /* ルートディレクトリの個数 */
    unsigned short datasec;     /* データ部先頭のセクタ番号 */
    unsigned short maxfat;      /* 総クラスタ数+1 */
    unsigned short dirsec;      /* ルートディレクトリの先頭セクタ番号 */
    int driver;                 /* デバイスドライバへのポインタ */
    unsigned char id;           /* メディアバイト */
    unsigned char flg;          /* dpb使用フラグ */
                                /* -1のときアクセスなし */
    struct _dpbptr *next;       /* 次のdpbへのポインタ */
    unsigned short dirfat;      /* カレントディレクトリのクラスタ番号 */
                                /* (0のときはルートディレクトリ) */
    char dirbuf[64];            /* カレントディレクトリの文字バッファ */
};
```

_dos_getdpb関数はDOSコール0xFF32を発行することによって処理される。

戻り値——成功した場合は 0 を返し，失敗した場合は負の値を返す。

規　格——*XC*

_dos_getenv

用　　途——環境変数を取得する。

書　　式——#include <sys/dos.h>

```
int _dos_getenv (const char *name, const char *env,
                 char *buff);
```

解　　説——_dos_getenv 関数は *env* で指定した環境から *name* で指定した**環境変数**検索し，その値を *buff* へ格納する。*env* に 0 を設定すると，親プロセスの環境を対象とする。

_dos_getenv 関数は DOS コール 0xFF53(0xFF83) を発行することによって処理される。

戻 り 値——失敗した場合は負の値を返す。

注　　意——*buff* は 256 バイト以上の領域を指していなければならない。

互 換 性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格——*XC*

関連項目——_dos_setenv

_dos_getfcb

用　途――ファイルコントロールブロック(FCB)を取得する。

書　式――#include <sys/dos.h>

```
union _fcb *_dos_getfcb (int fildes);
```

解　説――_dos_getfcb 関数は *fildes* で指定したファイルのファイルコントロールブロック(FCB)を取得する。戻り値の_fcb 共用体は次のとおり。

```
union _fcb {
    struct {
        unsigned char dupcnt;     /* DUP カウント */
        unsigned char devattr;    /* デバイス属性 */
        void *deventry;           /* デバイスエントリ */
        char nouse_1[8];          /* 未使用 */
        unsigned char openmode;   /* オープンモード */
        char nouse_2[21];         /* 未使用 */
        char name1[8];            /* ファイル名 1 */
        char ext[3];              /* 拡張子 */
        char nouse_3;             /* 未使用 */
        char name2[10];           /* ファイル名 2 */
        char nouse_4[38];         /* 未使用 */
    } chr;                        /* キャラクタデバイス */
    struct {
        unsigned char dupcnt;     /* DUP カウント */
        unsigned char physdrv;    /* 物理ドライブ番号 */
        void *deventry;           /* デバイスエントリ */
        unsigned int fileptr;     /* ファイルポインタ */
        unsigned int exclptr;     /* 排他制御情報へのポインタ */
        unsigned char openmode;   /* オープンモード */
        unsigned char entryidx;   /* ディレクトリエントリのセクタ内位置 */
        unsigned char clustidx;   /* アクセス中のクラスタ内のセクタ位置 */
        char nouse_2;             /* 不明 */
        unsigned short acluster;  /* アクセス中のクラスタ番号 */
        unsigned int asector;     /* アクセス中のセクタ番号 */
        void *iobuf;              /* I/O バッファの先頭 */
        unsigned long dirsec;     /* ディレクトリエントリのセクタ番号 */
        unsigned int fptrmax;     /* ファイルポインタの上限 */
        char name1[8];            /* ファイル名 1 */
        char ext[3];              /* 拡張子 */
        unsigned char attr;       /* ファイル属性 */
        char name2[10];           /* ファイル名 2 */
        unsigned short time;      /* 修正時刻 */
        unsigned short date;      /* 修正日付 */
        unsigned short fatno;     /* 先頭 FAT 番号 */
```

```
        unsigned long size;         /* ファイルサイズ */
        char nouse_4[28];           /* 未使用 */
    } blk;                          /* ブロックデバイス */
};
```

_dos_getfcb 関数は DOS コール 0xFF7C(0xFFAC) を発行することによって処理される。

戻り値——成功した場合は指定したファイルのファイルコントロールブロックへのポインタを返し，失敗した場合は負の値を返す。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——***Project LIBC Group***

_dos_getpdb

用　途——現在のプロセスのプロセス管理ポインタを求める。

書　式——#include <sys/dos.h>

```
struct _psp *_dos_getpdb (void);
```

解　説——_dos_getpdb 関数は現在のプロセスの**プロセス管理ポインタ**を求める。プロセス管理ポインタはプログラムの先頭アドレス−0xF0 のアドレスであり，**Human68k** の管理するメモリブロックの先頭+0x10 に位置する。戻り値の_psp 構造体は次のとおり。

```
struct _psp {
    char *env;                      /* 環境アドレス */
    void *exit;                     /* 終了時の戻りアドレス */
    void *ctrlc;                    /* CTRL+C アボート時の戻りアドレス */
    void *errexit;                  /* エラーアボート時の戻りアドレス */
    char *comline;                  /* プロセスのコマンドラインのアドレス */
    unsigned char handle[12];       /* プロセスのファイルハンドラの使用状況 */
    void *bss;                      /* bss の先頭アドレス */
    void *heap;                     /* heap の先頭アドレス */
    void *stack;                    /* 初期スタックアドレス */
    void *usp;                      /* 親プロセスの USP レジスタの値 */
    void *ssp;                      /* 親プロセスの SSP レジスタの値 */
    unsigned short sr;              /* 親プロセスの SR レジスタの値 */
    unsigned short abort_sr;        /* アボート時の SR レジスタの値 */
    void *abort_ssp;                /* アボート時の SSP レジスタの値 */
    void *trap10;                   /* tarp #10 のベクタアドレス */
    void *trap11;                   /* tarp #11 のベクタアドレス */
    void *trap12;                   /* tarp #12 のベクタアドレス */
    void *trap13;                   /* tarp #13 のベクタアドレス */
    void *trap14;                   /* tarp #14 のベクタアドレス */
    unsigned int osflg;             /* 0 で親プロセスあり，1 で OS から起動 */
    unsigned char reserve_1[28];    /* 未使用 */
    char exe_path[68];              /* exec されたファイルのパス名 */
    char exe_name[24];              /* exec されたファイル名 */
    char reserve_2[36];             /* 未使用 */
};
```

_dos_getpdb 関数は DOS コール 0xFF51(0xFF81) を発行することによって処理される。

戻り値——現在のプロセスの *pdbaddr* を返す。

互 換 性——**Human68k** ver.3ではDOSコールの番号が変更(+0x30)されたことに注意すること。

規　　格——*XC*

関連項目——_dos_setpdb

_dos_gets

用　途——文字列を入力する。

書　式——#include <sys/dos.h>

```
int _dos_gets (struct _inpptr *inpptr);
```

解　説——_dos_gets 関数は CR/LF コード (改行コード) が入力されるまで文字列を入力する。このとき，*inpptr* で指定した _inpptr 構造体のメンバ max に設定されている最大入力文字数に達した場合は警告音を出力するが，中断はしない。また，CR/LF コード (改行コード) が入力された場合は，バッファに書き込む前に null 文字に置換される。*inpptr* に指定する _inpptr 構造体は次のとおり。

```
struct _inpptr {
    unsigned char max;    /* 最大入力文字数 */
    unsigned char length; /* 実際の入力文字数 */
    char buffer[256];     /* 入力バッファ */
};
```

ブレークチェックで検査される文字には CTRL+C (ブレーク)，CTRL+S(表示の一時停止)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー停止) がある。

_dos_gets 関数は DOS コール 0xFF0A を発行することによって処理される。

戻り値——最後の null 文字を除く入力文字数を返す。

注　意——*inpptr* が示す _inpptr 構造体の buffer へ入力文字列を格納する。また _inpptr 構造体の length には，実際に入力が行われた文字数が格納される。

関数の戻り値は最後の CR/LF コードを含まない入力文字数である。

規　格——*XC*

関連項目——_dos_fgets

G

_dos_getss

用　　途――文字列を入力する。

書　　式――#include <sys/dos.h>

```
int _dos_getss (struct _inpptr *inpptr);
```

解　　説――_dos_getss関数は，入力データ中にCR/LFコード(改行コード)が現れるまで文字列を入力する。入力中はブレークチェックを行わず，VOID/NEWLINEが入力されても改行しない。

入力された文字列は*inpptr*で指定した_inpptr構造体のメンバlengthおよびbufferへ格納される。このとき，*inpptr*で指定した_inpptr構造体のメンバmaxに設定されている最大入力文字数に達した場合は警告音を出力するが，中断はしない。*inpptr*に指定する_inpptr構造体は次のとおり。

```
struct _inpptr {
    unsigned char max;     /* 最大入力文字数 */
    unsigned char length; /* 実際の入力文字数 */
    char buffer[256];      /* 入力バッファ */
};
```

_dos_getss関数はDOSコール0xFF1Aを発行することによって処理される。

戻 り 値――*inpptr*が示す_inpptr構造体のbufferへ入力文字列を格納する。また，_inpptr構造体のlengthには実際に入力された文字数が格納される。

関数の戻り値は，最後のCR/LFコードを含まない入力文字数である。

規　　格――*XC*

関連項目――_dos_fgets，_dos_gets

_dos_gettim2

用　　途——現在の時刻を取得する。

書　　式——

```
#include <sys/dos.h>
int _dos_gettim2 (void);
```

解　　説——_dos_gettim2 関数は現在の時刻を返す。

戻り値のビットフィールドの内容は次のとおり。

00000000 000hhhhh 00mmmmmm 00ssssss

- ssssss　　秒 (0 ～ 59)
- mmmmmm　　分 (0 ～ 59)
- hhhhh　　時 (0 ～ 23)

_dos_gettim2 関数は DOS コール 0xFF27 を発行することによって処理される。

戻 り 値——現在の時刻を返す。

規　　格——*XC*

関連項目——_dos_getdate, _dos_gettime, _dos_setdate, _dos_settim2, _dos_settime

G

_dos_gettime

用　　途——現在の時刻を取得する。

書　　式——#include <sys/dos.h>

```
int _dos_gettime (void);
```

解　　説——_dos_gettime 関数は現在の時刻を返す。

戻り値のビットフィールドの内容は次のとおり。

00000000 00000000 hhhhhmmm mmmsssss

- sssss　秒 (0 ～ 29) (実際の値は 2 を乗じる)
- mmmmmm　分 (0 ～ 59)
- hhhhh　時 (0 ～ 23)

_dos_gettime 関数は DOS コール 0xFF2C を発行することによって処理される。

戻 り 値——現在の時刻を返す。

規　　格——*XC*

関連項目——_dos_getdate, _dos_gettim2, _dos_setdate, _dos_settim2, _dos_settime

_dos_hendsp

用　途——漢字変換行をコントロールする。

書　式——

```
#include <sys/dos.h>
void _dos_hendspic (int position);
int  _dos_hendspio (void);
int  _dos_hendspip (int position, const char *message);
int  _dos_hendspir (int position, const char *message);
void _dos_hendspmc (void);
int  _dos_hendspmo (void);
int  _dos_hendspmp (int position, const char *message);
int  _dos_hendspmr (int position, const char *message);
void _dos_hendspsc (void);
int  _dos_hendspso (void);
int  _dos_hendspsp (int position, const char *message);
int  _dos_hendspsr (int position, const char *message);
```

解　説——_dos_hendspic 関数は *position* 以降の変換バッファの内容を変換前の状態へ戻す。

_dos_hendspio 関数は漢字変換ウィンドウをオープンする。

_dos_hendspip 関数は *position* で指定した位置から *message* の内容をノーマル属性で表示する。

_dos_hendspir 関数は *position* で指定した位置から *message* の内容をリバース属性で表示する。

_dos_hendspmc 関数はモード表示ウィンドウをクローズする。

_dos_hendspmo 関数はモード表示ウィンドウをオープンする。

_dos_hendspmp 関数は *position* で指定した位置から *message* の内容をノーマル属性で表示する。

_dos_hendspmr 関数は *position* で指定した位置から *message* の内容をリバース属性で表示する。

_dos_hendspsc 関数は候補ウィンドウをクローズする。

_dos_hendspso 関数は候補ウィンドウをオープンする。

_dos_hendspsp 関数は *position* で指定した位置から *message* の内容をノーマル属性で表示する。

_dos_hendspsr 関数は *position* で指定した位置から *message* の内容をリバース属性で表示する。

これらの関数は DOS コール 0xFF18 を発行することによって処理される。

戻り値——_dos_hendspic 関数，_dos_hendspmc 関数，_dos_hendspsc 関数には戻り値はない。

_dos_hendspio 関数は漢字変換ウィンドウの最大表示文字数を返す。

_dos_hendspip 関数と _dos_hendspir 関数は漢字変換ウィンドウ内の次の表示位置を返す。

_dos_hendspmo 関数はモード表示ウィンドウの最大表示文字数を返す。

_dos_hendspmp 関数と _dos_hendspmr 関数はモード表示ウィンドウ内の次の表示位置を返す。

_dos_hendspso 関数は候補ウィンドウの最大表示文字数を返す。

_dos_hendspsp 関数と _dos_hendspsr 関数は候補ウィンドウ内の次の表示位置を返す。

注　意——これらの関数は，本来漢字変換フロントエンドプロセッサのためのものなので，一般のアプリケーションプログラムから使用するべきではない。

規　格——*XC*

_dos_importlnenv

用　途——lndrv の管理情報へのポインタを返す。

書　式——#include <sys/dos.h>

```
int *_dos_importlnenv (void);
```

解　説——_dos_importlnenv 関数は lndrv が常駐しているかどうかを調べ，常駐していればその管理情報へのポインタを返す。

_dos_importlnenv 関数は Human68k の DOS コールではない。

戻り値——lndrv が常駐している場合は lndrv の管理情報へのポインタを返し，常駐していない場合は NULL を返す。

規　格——*Project LIBC Group*

関連項目——_dos_lfiles, _dos_readlink, _dos_symlink

_dos_indosflg

用　途――**Human68k** のワークフラグ INDOS_FLG のアドレスを取得する。

書　式――#include <sys/dos.h>

```
struct _indos *_dos_indosflg (void);
```

解　説――_dos_indosflg 関数は，**Human68k** の内部ワークエリアの INDOS_FLG のアドレスを返す。戻り値の_indos 構造体は次のとおり。

```
struct _indos {
    unsigned short indosf;       /* OS実行フラグ */
    unsigned char doscmd;        /* OS実行中ファンクション番号 */
    unsigned char fat_flg;       /* FAT検索モード (0＝標準, 0≠先頭から) */
    unsigned short retry_count; /* I/O リトライ数 (標準で3回) */
    unsigned short retry_time;  /* I/O リトライ待ち時間 (標準で100(1秒)) */
    unsigned short verifyf;      /* ベリファイモード (0＝オフ, 0≠オン) */
    unsigned char breakf;        /* ブレークモード (0＝オフ, 1＝オン) */
    unsigned char ctrlpf;        /* CTRL+P モード (0＝オフ, 0≠オン) */
    unsigned char reserved;      /* システム予約 */
    unsigned char wkcurdrv;      /* カレントドライブ (A＝0) */
};
```

_dos_indosflg 関数は DOS コール 0xFFF5 を発行することによって処理される。

戻り値――**Human68k** の内部ワークエリアの INDOS_FLG のアドレスを返す。

注　意――返されるアドレスはスーパーバイザ領域なので，ユーザモードではアクセスできない。また，このアドレス以降は **Human68k** にとって重要なワークエリアなので，書き込みは絶対に行わないこと。

規　格――*XC*

_dos_inkey

用　途――キーボードから１文字入力する。

書　式――#include <sys/dos.h>

```
int _dos_inkey (void);
```

解　説――_dos_inkey 関数はキーが押されるまで待ち，押されたキーコードを返す。このときブレークチェックは行わない。

_dos_inkey 関数は DOS コール 0xFF07 を発行することによって処理される。

戻り値――入力されたキーコードを返す。

注　意――ブレークチェックは行われないので，ブレークチェックが必要な場合は_dos_getc 関数を使用すること。

規　格――*XC*

関連項目――_dos_getc，_dos_getchar

_dos_inpout

用　　途――コンソールの直接入出力を行う。

書　　式――#include <sys/dos.h>

```
int _dos_inpout (int code);
```

解　　説――_dos_inpout 関数はコンソールに対して *code* で指定した処理を行う。

code に 0xFF が指定された場合はキー入力を行う。ただしキー入力がなくても入力を待たない。0xFE が指定された場合はキーセンスを行い，入力したキーコードを返す。それ以外の場合は *code* で指定した文字を出力する。

_dos_inpout 関数は DOS コール 0xFF06 を発行することによって処理される。

戻 り 値――*code* が 0xFF または 0xFE の場合は入力されたキーコードを返し，入力がない場合は 0 を返す。なお，文字を出力した場合は無条件に 0 を返す。

注　　意――ブレークチェックは行われない。

規　　格――*XC*

関連項目――_dos_getc，_dos_getchar，_dos_inkey，_dos_putchar

_dos_intvcg

用　途——割り込みベクタを取得する。

書　式——#include <sys/dos.h>

```
void *_dos_intvcg (int intno);
```

解　説——_dos_intvcg 関数は *intno* で指定した割り込みベクタの処理アドレスを取得する。

intno に指定できる値は次のとおり。

- 0x00 ～ 0xFF　割り込み処理
- 0x100 ～ 0x1FF　IOCS コール
- 0xFF00 ～ 0xFFFF　DOS コール

_dos_intvcg 関数は DOS コール 0xFF35 を発行することによって処理される。

戻り値——*intno* で指定したベクタの処理アドレスを返す。

規　格——*XC*

関連項目——_dos_intvcs

_dos_intvcs

用　　途——割り込みベクタを設定する。

書　　式——#include <sys/dos.h>

```
void *_dos_intvcs (int intno, const void *jobaddr);
```

解　　説——_dos_intvcs 関数は *intno* で指定した割り込みベクタに，*jobaddr* で指定した処理アドレスを設定する。

intno に指定できる値は次のとおり。

- 0x00 ～ 0xFF　　割り込み処理
- 0x100 ～ 0x1FF　　IOCS コール
- 0xFF00 ～ 0xFFFF　　DOS コール

_dos_intvcs 関数は DOS コール 0xFF25 を発行することによって処理される。

戻 り 値——設定変更前のベクタの処理アドレスを返す。

規　　格——*XC*

関連項目——_dos_intvcg

_dos_ioctrl

用　途――デバイスドライバを直接制御する。

書　式――

```
#include <sys/dos.h>
int _dos_ioctrlgt (int fildes);
int _dos_ioctrlst (int fildes, int data);
int _dos_ioctrlrh (int fildes, char *buff, int size);
int _dos_ioctrlwh (int fildes, char *buff, int size);
int _dos_ioctrlrd (int drive, char *buff, int size);
int _dos_ioctrlwd (int drive, char *buff, int size);
int _dos_ioctrlis (int fildes);
int _dos_ioctrlos (int fildes);
int _dos_ioctrldvctl (int drive, int f_code, char *buff);
int _dos_ioctrldvgt (int drive);
int _dos_ioctrlfdctl (int fildes, int f_code, char *buff);
int _dos_ioctrlfdgt (int fildes);
int _dos_ioctrlrtset (int count, int time);
```

解　説――_dos_ioctrlgt 関数は，*fildes*で指定したファイルハンドルのデバイス情報を調べる。この関数はデバイスドライバによる直接入出力を行うときに使用する。

_dos_ioctrlst 関数は *fildes*で指定したファイルハンドルのデバイス情報を *data* に設定する。*fildes*で指定するファイルハンドルは，キャラクタデバイスに対してオープンされたファイルハンドルでなければならない。*data*に設定できるのはビット 5 の RAW モードの設定のみであり，設定可能な値は次のとおり。

- 0x00　RAW モードに設定
- 0x20　COOKED モードに設定

_dos_ioctrlrh 関数は *fildes*で指定したファイルハンドルから，*size*で指定したバイト数分 *buff*へ読み込む。この関数はデバイスドライバによる直接入力を行うときに使用する。

_dos_ioctrlwh 関数は *fildes*で指定したファイルハンドルへ，*size*で指定したバイト数分 *buff*から書き出す。この関数はデバイスドライバによる直接出力を行うときに使用する。

_dos_ioctrlrd 関数は *drive*で指定したドライブのデバイスドライバから，*size*で指定したバイト数分 *buff*へ読み込む。この関数はデバイスドライバによる直接入力を行うときに使用する。

_dos_ioctrlwd 関数は *drive* で指定したドライブのデバイスドライバへ，*size* で指定したバイト数分 *buff* から書き出す。この関数はデバイスドライバによる直接出力を行うときに使用する。

_dos_ioctrlis 関数は *fildes* で指定したファイルハンドルの入力ステータスを調べる。

_dos_ioctrlos 関数は *fildes* で指定したファイルハンドルの出力ステータスを調べる。

_dos_ioctrldvctl 関数は *drive* で指定したドライブのデバイスドライバに対して *f_code* を指定して，デバイスドライバによる特殊コントロールを行う。*buff* にはデバイスドライバとの情報交換用の領域を指定する。この領域のサイズはデバイスドライバに依存する。また，*f_code* で指定される処理内容についてもデバイスドライバに依存する。

_dos_ioctrldvgt 関数は *drive* で指定したドライブがローカルなのかリモートなのかを調べる。

_dos_ioctrlfdctl 関数は *fildes* で指定したファイルハンドルに対して *f_code* を指定してデバイスドライバによる特殊コントロールを行う。*buff* にはデバイスドライバとの情報交換用の領域を指定する。この領域のサイズはデバイスドライバに依存する。また，*f_code* で指定される処理内容についてもデバイスドライバに依存する。

_dos_ioctrlfdgt 関数は *fildes* で指定したファイルがローカルなのかリモートなのかを調べる。

_dos_ioctrlrtset 関数はリトライ回数を *count* に，リトライ間隔を *time* に設定する。デフォルトの値はリトライ回数が3で，リトライ間隔は100(1秒) である。

これらの関数はDOSコール 0xFF44 を発行することによって処理される。

戻り値——_dos_ioctrlgt 関数はデバイス情報を返す。戻り値のデバイス属性は次のとおり。

- ビット 0　標準入力デバイス (CON)
- ビット 1　標準出力デバイス (CON)
- ビット 2　NUL デバイス
- ビット 3　CLOCK デバイス
- ビット 5　RAW モードかどうか (0 ならば COOKED モード)
- ビット 6　特殊 IOCTRL が可能かどうか (このビットが1でなければ，_dos_ioctrldvctl 関数，_dos_ioctrlfdctl 関数は使用できない)
- ビット 7　キャラクタデバイス (0 ならばブロックデバイスであり，ビット 0 ～ 4 はドライブ番号を表す)
- ビット 12　特殊ブロックデバイスのとき，デバイスがリモートかどうか (0 ならばローカル)

- ビット 13　特殊ブロックデバイス (0ならば通常のブロックデバイス)
- ビット 14　IOCTRL が可能かどうか (このビットが 1 でなければ, _dos_ioctrlrh 関数, _dos_ioctrlwh 関数, _dos_ioctrlrd 関数, _dos_ioctrlwd 関数は使用できない)
- その他　その他のビットはすべてシステム予約

ビット 7 が 0 の場合 (ブロックデバイスの場合), ビット 0 ～ 4 はドライブ番号 (0 ～ 25) を示す。

_dos_ioctrlst 関数は, 成功した場合は変更後のデバイス情報を返し, 失敗した場合は負の値を返す。

_dos_ioctrlrh 関数と _dos_ioctrlrd 関数は, 成功した場合は読み込んだバイト数を返し, 失敗した場合は負の値を返す。

_dos_ioctrlwh 関数と _dos_ioctrlwd 関数は, 成功した場合は書き込んだバイト数を返し, 失敗した場合は負の値を返す。

_dos_ioctrlis 関数はデバイスが入力可能ならば–1 を返し, 入力不可ならば 0 を返す。

_dos_ioctrlos 関数はデバイスが出力可能ならば–1 を返し, 出力不可ならば 0 を返す。

_dos_ioctrldvctl 関数と _dos_ioctrlfdctl 関数はデバイスドライバによって戻り値が異なるが, 失敗した場合は負の値を返す。

_dos_ioctrldvgt 関数と _dos_ioctrlfdgt 関数は, 成功した場合はデバイス情報を返し, 失敗した場合は負の値を返す。

_dos_ioctrlrtset 関数は, 失敗した場合は負の値を返す。

規　格—— *XC*

_dos_keeppr

用　　途——プロセスを常駐終了させる。

書　　式——#include <sys/dos.h>

```
void __volatiel _dos_keeppr (int prglen, int code);
```

解　　説——_dos_keeppr 関数は現在のプロセスを常駐終了させる。

*prglen*で指定したサイズ分 (プロセス管理ポインタの部分を除く) のプログラムが常駐し，呼び出した親プロセスには*code*が終了コードとして返される。このときオープンされているファイル (子プロセスがオープンしたファイルも含む) はすべてクローズされる。

_dos_keeppr 関数は DOS コール 0xFF31 を発行することによって処理される。

戻 り 値——なし。親プロセスには*code*が返される。

規　　格——*XC*

関連項目——_dos_exit, _dos_exit2

_dos_keyctrl

用　途——CON デバイスの直接入力制御を行う。

書　式——#include <sys/dos.h>

```
int _dos_k_keyinp (void);
int _dos_k_keysns (void);
int _dos_k_sftsns (void);
int _dos_k_keybit (int group);
int _dos_k_insmod (int mode);
```

解　説——_dos_k_keyinp 関数はキーボードから 1 文字入力する。

_dos_k_keysns 関数はキーボードからの入力を先読みする。ただし，“\0” が返された場合は入力がなかったことを示す。

_dos_k_sftsns 関数はシフトキーの押下げ状態を調べる。戻り値の下位 10 ビットには各シフトキーの押下げ状態が返されるが，LED のついているキーについてはLED の点灯状態を返す。戻り値の下位 10 ビットの内容は次のとおり。いずれも該当ビットが 1 の場合に押下されていることを示す。

10	9	8	7	6	5
全角	ひらがな	INS	CAPS	コード入力	ローマ字

4	3	2	1	0
かな	OPT.2	OPT.1	CTRL	SHIFT

_dos_k_insmod 関数は *mode* で指定した状態に INS キーの状態を設定する。*mode* に 0xFF を設定した場合は ON に，0x00 を設定した場合は OFF になる。

K

_dos_k_keybit 関数は *group* で指定したキーコードグループについて，その押下げ状態を調べる。キーコードグループと該当キーのビット位置の関係は次のとおり。縦軸がキーコードグループ，横軸がビット位置を表している。

	0	1	2	3	4	5	6	7
0	未定義	ESC	1 !	2 "	3 #	4 $	5 %	6 &
1	7 '	8 (	9)	0	- =	^	\ |	BS
2	TAB	Q	W	E	R	T	Y	U
3	I	O	P	@	[	CR	A	S
4	D	F	G	H	J	K	L	;
5	:	]	Z	X	C	V	B	N
6	M	, <	. >	/ ?	_	SPACE	HOME	DEL
7	ROLLUP	ROLLDN	UNDO	←	↑	→	↓	CLR
8	/	*	-	7	8	9	+	4
9	5	6	=	1	2	3	ENTER	0
A	,	.	記号	登録	HELP	XF1	XF2	XF3
B	XF4	XF5	かな	ローマ	コード	CAPS	INS	ひら
C	全角	BREAK	COPY	F1	F2	F3	F4	F5
D	F6	F7	F8	F9	F10	未定義	未定義	未定義
E	SHIFT	CTRL	OPT.1	OPT.2	未定義	未定義	未定義	未定義
F	未定義	未定義	未定義	未定義	未定義	未定義	未定義	未定義

これらの関数は DOS コール 0xFF24 を発行することによって処理される。

戻り値——_dos_k_keyinp 関数は入力した文字コードを返す。

_dos_k_keysns 関数は先読みした文字コードを返す。

_dos_k_sftsns 関数と _dos_k_keybit 関数は，キー状態をビットで返す。

_dos_k_insmod 関数には戻り値はない。

規　格——*XC*

_dos_keysns

用　途——キーの入力状態の検査を行う。

書　式——#include <sys/dos.h>

```
int _dos_keysns (void);
```

解　説——_dos_keysns 関数は標準入力の入力状態を検査する。このときブレークチェックが行われる。

_dos_keysns 関数は DOS コール 0xFF0B を発行することによって処理される。

戻り値——入力がある場合は−1 を返し，ない場合は 0 を返す。

注　意——ブレークチェックで検査される文字には，CTRL+C (ブレーク)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー停止) がある。

規　格——*XC*

関連項目——_dos_gets

_dos_kflush

用　途――入力バッファをフラッシュして，キーボード入力を行う。

書　式――

```
#include <sys/dos.h>
int _dos_kflushgc (void);
int _dos_kflushgp (void);
int _dos_kflushgs (struct _inpptr *inpptr);
int _dos_kflushin (void);
int _dos_kflushio (int code);
```

解　説――_dos_kflushgc 関数は，入力バッファをフラッシュして標準入力から1文字読み込む。このときブレークチェックが行われる。この処理は入力バッファをフラッシュして，_dos_getc 関数を実行するのと同じである。

_dos_kflushgp 関数は，入力バッファをフラッシュして標準入力から1文字読み込み，読み込んだ文字を標準出力へエコーバックする。このときブレークチェックが行われる。この処理は入力バッファをフラッシュして，_dos_getchar 関数を実行するのと同じである。

_dos_kflushgs 関数は，入力バッファをフラッシュして CR/LF コード (改行コード) が入力されるまで文字列を入力する。このとき *inpptr* で指定した _inpptr 構造体のメンバ max に設定されている最大入力文字数に達した場合は警告音を出力するが，中断はしない。また CR/LF コード (改行コード) が入力された場合は，バッファに書き込む前に null 文字に置換される。このときブレークチェックが行われる。この処理は入力バッファをフラッシュして，_dos_gets 関数を実行するのと同じである。*inpptr* に指定する _inpptr 構造体は次のとおり。

```
struct _inpptr {
    unsigned char max;     /* 最大入力文字数 */
    unsigned char length;  /* 実際の入力文字数 */
    char buffer[256];      /* 入力バッファ */
};
```

_dos_kflushin 関数は，入力バッファをフラッシュしてからキー入力があるまで待ち，入力された文字コードを返す。このときブレークチェックは行われない。この処理は入力バッファをフラッシュして，_dos_inkey 関数を実行するのと同じである。

_dos_kflushio 関数は，コンソールに対して *code* で指定した処理を行う。ただし，*code* に 0xFF が指定された場合はキー入力 (ただしキー入力がなくても入力を待たない) を行い，0xFE が指定された場合はキーセンスを行って，入力したキーコードを返す。それ以外の場合は *code* で指定した文字を出力する。このと

きブレークチェックは行われない。この処理は入力バッファをフラッシュして，_dos_inpout 関数を実行するのと同じである。

これらの関数は DOS コール 0xFF0C を発行することによって処理される。

戻り値――_dos_kflushgc 関数，_dos_kflushgp 関数は，読み込んだ文字コードを返す。

_dos_kflushgs 関数は最後の null 文字を除く入力文字数を返す。

_dos_kflushin 関数は入力された文字コードを返す。

_dos_kflushio 関数は，*code* が 0xFF または 0xFE の場合は入力されたキーコードを返し，入力がない場合は 0 を返す。また，文字を出力した場合は無条件に 0 を返す。

注　意――_dos_kflushgc 関数，_dos_kflushgp 関数，_dos_kflushgs 関数において，ブレークチェックで検査される文字には CTRL+C (ブレーク)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー停止) がある。

規　格――*XC*

関連項目――_dos_getc，_dos_getchar，_dos_gets，_dos_inkey，_dos_inpout

K

_dos_kill_pr

用　　途——自分自身のプロセスを削除する。

書　　式——#include <sys/dos.h>

```
int _dos_kill_pr (void);
```

解　　説——_dos_kill_pr 関数は自分自身のプロセスを削除する。

常駐終了していたプロセスの場合，同じプロセスID(タスク管理情報のメンバpsp_id) をもつすべてのスレッドを削除し，確保されていたメモリをすべて解放する。

_dos_kill_pr 関数を実行する場合は，自分でオープンしたファイルはすべてクローズし，書き換えたベクタがあるならばすべて元に戻した後で実行すること。**Human68k** はスレッドの削除とメモリの解放以外は行わない。

プロセス内でいくつかのスレッドを登録し実行した後で，_dos_exit 関数，_dos_exit2 関数でプロセスを終了する場合は，プロセス内でオープンしたスレッドのみを_dos_kill_pr 関数で削除してから，プロセスを終了すること。

_dos_kill_pr 関数はDOSコール 0xFFF9 を発行することによって処理される。

戻 り 値——失敗した場合は負の値を返す。

注　　意——メインのスレッドであるプロセスを削除した場合，その後の動作は保証されない。

規　　格——*XC*

関連項目——_dos_change_pr，_dos_get_pr，_dos_open_pr，_dos_send_pr，_dos_sleep_pr，_dos_suspend_pr，_dos_time_pr

_dos_lfiles

用　途——シンボリックリンクを処理せずにファイルを検索する。

書　式——#include <sys/dos.h>

```
int _dos_lfiles (struct _filbuf *buff,
                 const char *name, int atr);
```

解　説——_dos_lfiles 関数は *name* で指定したファイル名と *atr* で指定したファイル属性のファイルを検索し，*buff* へその情報をセットする。ただし，_dos_files 関数とは異なり，*name* がシンボリックリンクファイルを指しているのに，*file* のシンボリックリンクファイル自身の情報を検索する。つまり，シンボリックリンクを処理せずに検索することになる。

_dos_lfiles 関数は，*file* にシンボリックリンクファイル以外を指定したか，**lndrv** が常駐していない場合には _dos_files 関数と同じである。*buff* に指定する _filbuf 構造体は次のとおり。

```
struct _filbuf {
    unsigned char searchatr; /* 検索するファイル属性 */
    unsigned char driveno;   /* ドライブ番号 */
    unsigned long dirsec;    /* ディレクトリエントリのセクタ番号 */
    unsigned short dirlft;   /* ディレクトリエントリの残りのセクタ数 */
    unsigned short dirpos;   /* ディレクトリエントリ内の位置 */
    char filename[8];        /* ファイルのノード名 */
    char ext[3];             /* ファイルの拡張子名 */
    unsigned char atr;       /* ファイル属性 */
    unsigned short time;     /* ファイル時刻 */
    unsigned short date;     /* ファイル日時 */
    unsigned int filelen;    /* ファイル長 */
    char name[23];           /* ファイル名 */
};
```

_dos_lfiles 関数は **Human68k** の DOS コールではない。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

L

注　意——_filbuf 構造体の searchatr から ext までのメンバは **Human68k** が使用するので，ユーザは変更してはならない。変更した場合，_dos_nfiles 関数が使用できなくなる。

_filbuf 構造体のメンバ dirpos は *name* にワイルドカードを指定したときにだけ正しく設定され，それ以外では 0xFFFF が設定される。また，dirpos が 0xFFFF の場合は，_dos_nfiles 関数による再検索は行えない。

_dos_lfiles 関数の動作はすべて **lndrv** に依存するため，**lndrv** 以外のシンボリックリンクドライバでは使用することができない。また，将来 **lndrv** が変更されても動作しなくなる可能性がある。

規　格——***Project LIBC Group***

関連項目——_dos_importlnenv，_dos_readlink，_dos_symlink

_dos_link

用　途――ハードリンクファイルを作成する。

書　式――
```
#include <sys/dos.h>
int _dos_link (const char *src, const char *dst);
```

解　説――_dos_link 関数は，*src* と同じファイルの実体を指すような**ハードリンク**ファイル *dst* を作成する。ただし，*dst* と同名のファイルがすでに存在する場合は失敗する。

ハードリンクするファイルは同一**ファイルシステム**上のファイルでなければならない。ハードリンクファイルの削除は，_dos_unlink 関数でのみ正常に実行することができる。

_dos_link 関数は **Human68k** の DOS コールではない。

戻り値――成功した場合は 0 を返し，失敗した場合は負の値を返す。

注　意――**Human68k** にはハードリンクの概念がないため，_dos_link 関数では実現するために非常に危険な方法を採用している。一般的には，_dos_link 関数が作成するハードリンクファイルは「複数ファイルが同じセクタを指している」という破損ファイルとして認識されてしまう。不用意な使用はファイルシステムの破壊になりかねないので注意すること。

規　格――*Project LIBC Group*

関連項目――_dos_unlink

L

_dos_lock

用　途——ファイルのロックを設定/解除する。

書　式——#include <sys/dos.h>

```
int _dos_lock (int fildes, int offset, int len);
int _dos_unlock (int fildes, int offset, int len);
```

解　説——_dos_lock関数は*fildes*で指定したファイルに対してロックを行い，他のプロセスからのファイルアクセスを禁止する。*fildes*にはロックしたいファイルのファイルハンドルを，*offset*にはファイル中のロックしたい部分の先頭からのオフセットを，*len*には*offset*からロックする長さをバイト単位で指定する。

_dos_unlock関数は*fildes*で指定したファイルのロックを解除する。*fildes*にはロックを解除したいファイルのファイルハンドルを，*offset*にはファイル中のロックを解除したい部分の先頭からのオフセットを，*len*には*offset*からロック解除する長さをバイト単位で指定する。

_dos_lock関数と_dos_unlock関数により，ファイルアクセスの排他制御を行い，他のプロセスが勝手にファイルを更新できないようにすることができる。

これらの関数はDOSコール0xFF5C(0xFF8C)を発行することによって処理される。

戻り値——失敗した場合は負の値を返す。

互換性——**Human68k** ver.3ではDOSコールの番号が変更(+0x30)されたことに注意すること。

規　格——*XC*

_dos_maketmp

用　途——指定したパスにテンポラリファイルを作成する。

書　式——#include <sys/dos.h>

```
int _dos_maketmp (const char *file, int atr);
```

解　説——_dos_maketmp 関数は *file* で指定したパス名/ファイル名の**テンポラリファイル**を *atr* で指定した属性で作成する。

file には，テンポラリファイルを作成するパス名とテンポラリファイルのファイル名を指定する。テンポラリファイルのファイル名は次のような形の文字列を設定する。

```
file????.txt
```

この “????” の部分は **Human68k** によって 0000 ～ 9999 までの数字に変換される。このとき “????” の代わりに数字を指定しておくと，その数字から検索され，ファイル名が決定する。*atr* に指定するファイル属性は_dos_create 関数と同じである。

_dos_maketmp 関数は DOS コール 0xFF5A(0xFF8A) を発行することによって処理される。

戻り値——成功した場合は作成したテンポラリファイルのファイルハンドルを返し，失敗した場合は負の値を返す。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

関連項目——_dos_create

M

_dos_malloc

用　途——メモリを確保する。

書　式——#include <sys/dos.h>

```
#include <sys/dos.h>
void *_dos_malloc (int size);
```

解　説——_dos_malloc 関数は *size* バイト分のメモリを確保する。

_dos_malloc 関数は DOS コール 0xFF48 を発行することによって処理される。

戻り値——メモリが確保できた場合は確保した領域のアドレスを返し，確保できなかった場合は 0x81000000+最大バイト数を返す。また，完全に確保できるメモリが不足している場合は0x8200000?を返す。

注　意——*size* に 0x1000000 以上の値を指定した場合は，0xFFFFFF が指定されたとみなす。

規　格——***XC***

_dos_malloc2

用　途——メモリを指定した方法で確保する。

書　式——#include <sys/dos.h>

```
void *_dos_malloc2 (int mode, int size);
```

解　説——_dos_malloc2 関数は *mode* で指定した割りあてモードで，*size* バイト分のメモリを確保する。

mode に指定できる値は次のとおり。

- 0　メモリの下位から探す
- 1　指定サイズを満たすメモリブロックのうち，最小サイズのブロックを割りあてる
- 2　メモリの上位から探す

_dos_malloc2 関数は DOS コール 0xFF58(0xFF88) を発行することによって処理される。

M

戻り値——メモリが確保できた場合は確保した領域のアドレスを返し，確保できなかった場合は 0x81000000+最大バイト数を返す。また，完全に確保できるメモリが不足している場合は0x8200000?を返す。

注　意——*size* に 0x1000000 以上の値を指定した場合は，0xFFFFFF が指定されたとみなす。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

_dos_memcpy

用　　途――バスエラーが発生するかどうかをテストする。

書　　式――#include <sys/dos.h>

```
int _dos_memcpy (const void *s_addr, void *d_addr, int mode);
```

解　　説――_dos_memcpy 関数はバスエラーが発生するかどうかをテストする。何もマッピングされていない領域やスーパーバイザ領域など，アクセスしたときにバスエラーが発生する可能性のある領域を読み書きできるかどうか，あらかじめテストすることができる。*s_addr* には読み込む領域へのポインタを，*d_addr* には書き込む領域へのポインタを指定する。*mode* にはアクセスするときのバイトサイズを 1, 2, 4 のいずれかで指定する。これらはそれぞれバイト，ワード，ロングワードを示す。

_dos_memcpy 関数は DOS コール 0xFFF7 を発行することによって処理される。

戻り値――戻り値が 0 ならば，読み書きともに可能であることを示し，*s_addr* で指定したアドレスからデータを読み込み，*d_addr* で指定したアドレスへデータを書き込む。

戻り値が 1 ならば，*d_addr* で示したアドレスへの書き込みが失敗したことを示す。

戻り値が 2 ならば，*s_addr* で示したアドレスからの読み込みが失敗したことを示す。

戻り値が−1 ならば，*mode* が不正または *mode* に 2, 4 を指定したにもかかわらず，*s_addr* / *d_addr* が奇数アドレスであることを示す。

s_addr を先にテストするため，*s_addr* / *d_addr* 両方でバスエラーが発生した場合，戻り値は 2 となる。

注　　意――この DOS コールのエントリを，_dos_intvcs 関数によって変更してはならない。

規　　格――*XC*

_dos_mfree

用　途——メモリブロックを解放する。

書　式——
```
#include <sys/dos.h>
int _dos_mfree (void *memptr);
```

解　説——_dos_mfree 関数は *memptr* で指定したメモリブロックを解放する。

memptr には，_dos_malloc 関数および_dos_malloc2 関数によって確保した領域へのポインタを指定する。したがって，_dos_malloc 関数および_dos_malloc2 関数によって確保した領域以外のポインタを *memptr* に設定するとエラーとなる。

memptr に 0 を指定すると，自プロセスと子プロセスが確保したすべてのメモリブロックを解放する。

_dos_mfree 関数は DOS コール 0xFF49 を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。

規　格——*XC*

関連項目——_dos_malloc，_dos_malloc2

_dos_mkdir

用　　途――ディレクトリを作成する。

書　　式――#include <sys/dos.h>

```
int _dos_mkdir (const char *file);
```

解　　説――_dos_mkdir 関数は *file* で指定したディレクトリを作成する。

_dos_mkdir 関数は DOS コール 0xFF39 を発行することによって処理される。

戻 り 値――成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

規　　格――*XC*

関連項目――_dos_chdir，_dos_rmdir

_dos_move

用　途——ファイルを移動する。

書　式——#include <sys/dos.h>

```
int _dos_move (const char *oldfile, const char *newfile);
```

解　説——_dos_move 関数は *oldfile* で指定したファイル名を，*newfile* で指定したファイル名に変更する。

_dos_rename 関数とは異なり，異なるパス名を指定することによってファイルを移動することができる。ただし，異なるドライブ間にまたがる移動は行われない。

すでにオープンされているファイルに対して実行した場合はエラーとなる。

_dos_move 関数は DOS コール 0xFF56(0xFF86) を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

関連項目——_dos_rename

_dos_nameck

用　　途——ファイル名を解析する。

書　　式——#include <sys/dos.h>

```
int _dos_nameck (const char *file, struct _nameckbuf *buff);
```

解　　説——_dos_nameck 関数は *file* で指定したファイル名を解析し，その結果を *buff* で指定した領域へ格納する。*buff* に指定する _nameckbuf 構造体は次のとおり。

```
struct _nameckbuf {
    char drive[2]; /* ドライブ */
    char path[65]; /* パス名 (ルート='\', サブ='\subdir\') */
    char name[19]; /* ファイル名 */
    char ext[5];   /* 拡張子 */
};
```

_dos_nameck 関数は DOS コール 0xFF37 を発行することによって処理される。

戻 り 値——失敗した場合は負の値を返す。ただし 0xFF のときは，ファイルが指定されていない場合は 0xFF を，ワイルド指定がない場合は 0x00 を返す。それ以外の場合はワイルド指定があることを表す。

規　　格——*XC*

_dos_namests

用　　途――ファイル名を解析する。

書　　式――

```
#include <sys/dos.h>
int _dos_namests (const char *file, struct _namestbuf *buff);
```

解　　説――_dos_namests 関数は *file* で指定したファイル名の解析を行い，その結果を解析結果を *buff* で指定した領域へ格納する。*buff* に指定する _namestbuf 構造体は次のとおり。

```
struct _namestbuf {
    unsigned char flg;    /* 0= ワイルドカードなし 0xFF= ファイル指定なし */
    unsigned char drive;  /* ドライブ番号 (A=0, B=1, ...) */
    char path[65];        /* パス名 (ルート='\', サブ='\subdir\') */
    char name1[8];        /* ファイル名 (先頭 8 文字) */
    char ext[3];          /* 拡張子 */
    char name2[10];       /* ファイル名 (9 文字目以降) */
};
```

_dos_namests 関数は DOS コール 0xFF29 を発行することによって処理される。

戻 り 値――成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。

規　　格――*XC*

_dos_newfile

用　　途――ファイルを新規に作成する。

書　　式――#include <sys/dos.h>

```
int _dos_newfile (const char *file, dosmode_t atr);
```

解　　説――_dos_newfile 関数は，*file* で示したファイルを *atr* のファイル属性で新規に作成する。_dos_create 関数と異なり，すでに存在する場合はエラーとなる。

atr に指定できる値は次のとおり。

- ビット 0　読み込み専用
- ビット 1　隠しファイル
- ビット 2　システムファイル
- ビット 3　ボリュームラベル
- ビット 4　ディレクトリ
- ビット 5　通常のファイル

_dos_newfile 関数は DOS コール 0xFF5B(0xFF8B) を発行することによって処理される。

戻 り 値――成功した場合は作成されたファイルハンドルを返し，失敗した場合は負の値を返す。ただし，ファイルがすでに存在していた場合は−80 を返す。

互 換 性――**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格――*XC*

関連項目――_dos_create

_dos_nfiles

用　　途——_dos_files 関数で検索された次のファイルを検索する。

書　　式——#include <sys/dos.h>

```
int _dos_nfiles (struct _filebuf *buff);
```

解　　説——_dos_nfiles 関数は_dos_files 関数で検索された次のファイルを検索する。

*buff*には_dos_files 関数で設定された_filbuf 構造体へのポインタを指定すること。*buff*に指定する_filbuf 構造体は次のとおり。

```
struct _filbuf {
    unsigned char searchatr; /* 検索するファイル属性 */
    unsigned char driveno;   /* ドライブ番号 */
    unsigned long dirsec;    /* ディレクトリエントリのセクタ番号 */
    unsigned short dirlft;   /* ディレクトリエントリの残りのセクタ数 */
    unsigned short dirpos;   /* ディレクトリエントリ内の位置 */
    char filename[8];        /* ファイルのノード名 */
    char ext[3];             /* ファイルの拡張子名 */
    unsigned char atr;       /* ファイル属性 */
    unsigned short time;     /* ファイル時刻 */
    unsigned short date;     /* ファイル日時 */
    unsigned int filelen;    /* ファイル長 */
    char name[23];           /* ファイル名 */
};
```

_dos_nfiles 関数は DOS コール 0xFF4F を発行することによって処理される。

戻 り 値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

注　　意——_filbuf 構造体の searchatr から ext までのメンバは **Human68k** が使用するので，ユーザは変更してはならない。変更した場合，_dos_nfiles 関数が使用できなくなる。

規　　格——*XC*

関連項目——_dos_files

N

_dos_open

用　途――ファイルをオープンする。

書　式――#include <sys/dos.h>

```
int _dos_open (const char *file, int mode);
```

解　説――_dos_open 関数は *file* で指定したファイルを，*mode* で指定したモードでオープンする。

mode に指定できる値は次のとおり。

- 0x000　読み込み
- 0x001　書き込み
- 0x002　読み込み，書き込み
- 0x100　読み込み (辞書用特殊ハンドル)
- 0x101　書き込み (辞書用特殊ハンドル)
- 0x102　読み込み，書き込み (辞書用特殊ハンドル)

_dos_open 関数は DOS コール 0xFF3D を発行することによって処理される。

戻り値――成功した場合はオープンしたファイルハンドルを返し，失敗した場合は負の値を返す。

注　意――*mode* に 0x100 ～ 0x102 を指定すると，辞書用の特殊ハンドルを作成するが，この特殊ハンドルはユーザが使用してはならない。

規　格――*XC*

_dos_open_pr

用　途——バックグラウンドプロセスを登録する。

書　式——#include <sys/dos.h>

```
int _dos_open_pr (const char *name, int counter,
                  int usp, int ssp, int sr, int pc,
                  struct _prcctrl *buff, long sleep_time);
```

解　説——_dos_open_pr 関数は**バックグラウンドプロセス**を登録する。登録したスレッドはSLEEP 状態となる。

*name*にはスレッドの名前を示す 15 文字以内の文字列へのポインタを指定するが，もし *name*で指定したスレッドがすでに存在する場合はエラーとなる。

*counter*にはタスクを 1 回実行するために，タスク切り替えの**タイマ割り込み**を何回カウントするのかを指定する。*counter*に指定できる値は 2 ～ 255 であり，0 や 1 が指定された場合は 2 が指定されたものとして扱う。

usp，*ssp*，*sr*の値はそれぞれタスクを実行するときに渡されるレジスタの初期値である。システム用のスタックには 6K バイトが必要である。また，*sr*は 0 または 0x2000 を指定し，実行するプロセスがユーザモードとスーパーバイザモードのどちらで実行されるのかを指定する。0x2000 が指定された場合，実行プロセスは**スーパーバイザモード**で実行される。その他のレジスタはすべて 0 に初期化されプロセスに渡される。

*buff*には**タスク間通信**用のバッファへのポインタを指定する。*buff*に指定する _prcctrl 構造体の内容は次のとおり。

```
struct _prcctrl {
    long length;              /* データバッファの長さ */
    unsigned char *buf_ptr; /* データバッファへのポインタ */
    unsigned short command; /* コマンドバッファ */
    unsigned short your_id; /* 相手の ID のバッファ (-1 で通信許可) */
};
```

*sleep_time*には起動するまでの待ち時間をミリ秒単位で指定する。0 を指定した場合は永久に SLEEP する。

_dos_open_pr 関数により，バックグラウンドプロセスを登録した場合には _dos_keeppr 関数によって常駐終了する。また，スレッドを削除する場合は _dos_kill_pr 関数を使用する。

_dos_open_pr 関数は DOS コール 0xFFF8 を発行することによって処理される。

O

戻り値——スレッドが登録できた場合はスレッドのIDを返し，失敗した場合は負の値を返す。ただし−27が返された場合は，*name*で指定した名前のスレッドがすでに起動されていたことを示し，−29が返された場合はスレッド数が多すぎて起動できなかったことを示す。

規　格——*XC*

関連項目——_dos_change_pr，_dos_get_pr，_dos_kill_pr，_dos_send_pr，_dos_sleep_pr，_dos_suspend_pr，_dos_time_pr

_dos_print

用　　途——文字列を表示する。

書　　式——#include <sys/dos.h>

```
void _dos_print (const char *msgptr);
```

解　　説——_dos_print 関数は *msgptr* で指定した文字列を表示する。このときブレークチェックが行われる。

_dos_print 関数は DOS コール 0xFF09 を発行することによって処理される。

戻 り 値——なし。

注　　意——表示する文字列は null 文字で終端していなくてはならない。ブレークチェックで検査される文字には，CTRL+C (ブレーク)，CTRL+S (表示の一時停止)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー停止) がある。

規　　格——*XC*

関連項目——_dos_gets

_dos_prnout

用　　途——プリンタに1文字出力する。

書　　式——#include <sys/dos.h>

```
void _dos_prnout (int code);
```

解　　説——_dos_prnout 関数は *code* で指定した1文字をプリンタへ出力する。このときブレークチェックが行われる。

_dos_prnout 関数は DOS コール 0xFF05 を発行することによって処理される。

戻 り 値——なし。

注　　意——ブレークチェックで検査される文字は CTRL+C (ブレーク) のみである。*code* に漢字コードを直接指定することはできない。この場合は，漢字コードの上位/下位の順で1バイトずつ出力すること。

規　　格——*XC*

関連項目——_dos_prnsns

_dos_prnsns

用　　途――プリンタへの出力が可能かどうかを調べる。

書　　式――#include <sys/dos.h>

```
int _dos_prnsns (void);
```

解　　説――_dos_prnsns 関数は，現在プリンタへの出力が可能であるかどうかを調べる。

_dos_prnsns 関数は DOS コール 0xFF11 を発行することによって処理される。

戻り値――プリンタへの出力が可能な場合は 0 以外を返し，不可能ならば 0 を返す。

規　　格――*XC*

P

_dos_pspset

用　途——プロセス管理情報を設定する。

書　式——#include <sys/dos.h>

```
void _dos_pspset (const struct _psp *pspaddr);
```

解　説——_dos_pspset 関数は *pspaddr* で指定したアドレスにプロセス管理情報を書き込む。*pspaddr* に指定する _psp 構造体は次のとおり。

```
struct _psp {
    char *env;                          /* 環境アドレス */
    void *exit;                         /* 終了時の戻りアドレス */
    void *ctrlc;                        /* CTRL+C 時の戻りアドレス */
    void *errexit;                      /* エラー時の戻りアドレス */
    char *comline;                      /* プロセスのコマンドラインのアドレス */
    unsigned char handle[12];           /* プロセスのファイルハンドラの使用状況 */
    void *bss;                          /* bss の先頭アドレス */
    void *heap;                         /* heap の先頭アドレス */
    void *stack;                        /* 初期スタックアドレス */
    void *usp;                          /* 親プロセスの USP レジスタの値 */
    void *ssp;                          /* 親プロセスの SSP レジスタの値 */
    unsigned short sr;                  /* 親プロセスの SR レジスタの値 */
    unsigned short abort_sr;            /* アボート時の SR レジスタの値 */
    void *abort_ssp;                    /* アボート時の SSP レジスタの値 */
    void *trap10;                       /* tarp #10 のベクタアドレス */
    void *trap11;                       /* tarp #11 のベクタアドレス */
    void *trap12;                       /* tarp #12 のベクタアドレス */
    void *trap13;                       /* tarp #13 のベクタアドレス */
    void *trap14;                       /* tarp #14 のベクタアドレス */
    unsigned int osflg;                 /* 0 で親プロセスあり, 1 で OS から起動 */
    unsigned char reserve_1[28];        /* 未使用 */
    char exe_path[68];                  /* exec されたファイルのバス名 */
    char exe_name[24];                  /* exec されたファイル名 */
    char reserve_2[36];                 /* 未使用 */
};
```

_dos_pspset 関数は DOS コール 0xFF26 を発行することによって処理される。

戻り値——なし。

注　　意——*pspaddr*は，**Human68k** のメモリ管理ポインタのアドレス+0x10 バイトを指定しなくてはならない。そのため_dos_malloc 関数で領域を確保し，そのアドレスを使用すること。なお，_psp 構造体のサイズは 240 バイトである。

規　　格——*XC*

関連項目——_dos_malloc

_dos_putchar

用　途——標準出力へ1文字出力する。

書　式——#include <sys/dos.h>

```
void _dos_putchar (int code);
```

解　説——_dos_putchar 関数は，*code*で指定した1文字を標準出力へ出力する。このときブレークチェックが行われる。

_dos_putchar 関数は DOS コール 0xFF02 を発行することによって処理される。

戻り値——なし。

注　意——ブレークチェックで検査される文字には，CTRL+C (ブレーク)，CTRL+S (表示の一時停止)，CTRL+P (ブレークエコー開始)，CTRL+N (ブレークエコー停止) がある。

規　格——*XC*

関連項目——_dos_getchar，_dos_inpout，_dos_fputc

_dos_read

用　　途── ファイルからデータを読み込む。

書　　式── #include <sys/dos.h>

```
int _dos_read (int fildes, char *buff, int size);
```

解　　説── _dos_read 関数は *fildes* で指定したファイルから，*size* バイト分を *buff* の指す領域に読み込む。

_dos_read 関数は DOS コール 0xFF3F を発行することによって処理される。

戻 り 値── 成功した場合は読み込んだバイト数を返し，失敗した場合は負の値を返す。ただし，読み込みの最中でファイルの終端に達した場合はそれまでに読み込んだバイト数を返し，読み込みを開始する前にすでにファイルの終端に達していた場合は 0 を返す。

規　　格── *XC*

関連項目── _dos_write

R

_dos_readlink

用　　途——シンボリックリンクのリンク先を調べる。

書　　式——
```
#include <sys/dos.h>
int _dos_readlink (const char *linkpath,
                   char *namebuf, int buflen);
```

解　　説——_dos_readlink 関数は，*linkpath*で指定したシンボリックリンクファイルが指しているリンク先ファイル(*linkpath*ファイルの中身)のパス名を調べ，その結果を*namebuf*で指定した*buflen*バイトの領域にコピーする。

_dos_readlink 関数は **Human68k** の DOS コールではない。

戻 り 値——成功した場合は場合は 0 を返し，失敗した場合は負の値を返す。

注　　意——_dos_readlink 関数の動作はすべて **lndrv** に依存するため，**lndrv** 以外のシンボリックリンクドライバでは使用することができない。また，将来 **lndrv** が変更されたりしても動作しなくなる可能性がある。

*namebuf*は最低でも*buflen*バイト以上の領域を指していなければならない。

規　　格——***Project LIBC Group***

関連項目——_dos_importlnenv，_dos_lfiles，_dos_symlink

_dos_rename

用　　途——ファイル名を変更する。

書　　式——

```
#include <sys/dos.h>
int _dos_rename (const char *oldfile, const char *newfile);
```

解　　説——_dos_rename 関数は *oldfile* で指定したファイル名を *newfile* で指定したファイル名に変更する。すでにオープンされているファイルに対して実行した場合はエラーとなる。

_dos_rename 関数は DOS コール 0xFF56(0xFF86) を発行することによって処理される。

戻 り 値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。通常は 0 が返るが，特殊デバイスドライバが対象の場合は 0 以外の正の値の場合もある。

互 換 性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格——*XC*

関連項目——_dos_move

R

_dos_retshell

用　　途——コマンドシェルにジャンプする。

書　　式——

```
#include <sys/dos.h>
void __volatile _dos_retshell (void);
```

解　　説——_dos_retshell 関数は，強制的にコマンドシェルに実行を移す。ただし，スタックの調整も割り込みベクタの復帰も行われないため，ユーザは使用してはならない。

_dos_retshell 関数は DOS コール 0xFFF0 を発行することによって処理される。

戻り値——なし。

規　　格——***Project LIBC Group***

_dos_rmdir

用　　途——ディレクトリを削除する。

書　　式——#include <sys/dos.h>

```
int _dos_rmdir (const char *file);
```

解　　説——_dos_rmdir関数は*file*で指定したディレクトリを削除する。

_dos_rmdir関数はDOSコール0xFF3Aを発行することによって処理される。

戻り値——成功した場合は0を含む正の値を返し，失敗した場合は負の値を返す。通常は0が返るが，特殊デバイスドライバが対象の場合は0以外の正の値の場合もある。

規　　格——*XC*

関連項目——_dos_chdir，_dos_mkdir

R

_dos_s_malloc

用　　途――メインのメモリ管理下からメモリブロックを確保する。

書　　式――

```
#include <sys/dos.h>
void *_dos_s_malloc (int mode, int size);
```

解　　説――_dos_s_malloc 関数は，*size* で指定したバイト数のメモリをメインのメモリ管理から確保し，そのポインタを返す。

mode に指定できる値は次のとおり。

- 0　メモリの下位から探す
- 1　指定サイズを満たすメモリブロックのうち，最小サイズのブロックを割りあてる
- 2　メモリの上位から探す

_dos_s_malloc 関数は DOS コール 0xFF7D(0xFFAD) を発行することによって処理される。

戻 り 値――メモリが確保できた場合は確保した領域のアドレスを返し，確保できなかった場合は 0x81000000+最大バイト数を返す。完全に確保できるメモリが不足している場合は0x8200000? を返す。

注　　意――_dos_s_malloc 関数は，通常のアプリケーションから使用してはならない。また，_dos_s_malloc 関数を使用するプログラムは OS から起動され，終了することができない (常駐プロセスのように_dos_exit 関数や_dos_exit2 関数を使用しない) プログラムでなくてはならない。

size に 0x1000000 以上の値を指定した場合は，0xFFFFFF が指定されたとみなす。

互 換 性――**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格――*XC*

関連項目――_dos_malloc，_dos_mfree，_dos_s_mfree，_dos_s_process

_dos_s_mfree

用　途——メインのメモリ管理下のメモリブロックを解放する。

書　式——#include <sys/dos.h>

```
int _dos_s_mfree (void *memptr);
```

解　説——_dos_s_mfree 関数は，*memptr*で指定したメインのメモリ管理下のメモリブロックを解放する。

*memptr*には_dos_s_malloc 関数で確保した領域へのポインタを指定する。したがって，それ以外のポインタを*memptr*に設定するとエラーとなる。

*memptr*が_dos_s_process 関数で指定しているアドレスで，かつそのスレッド ID がカレントスレッド ID ならば，自分自身を削除する。このとき内部のメモリ管理で常駐終了しているプロセスはメインのメモリ管理に連結される。

_dos_s_mfree 関数は DOS コール 0xFF7E(0xFFAE) を発行することによって処理される。

戻り値——親プロセスへは 0 が返される。

注　意——_dos_s_mfree 関数は，*memptr*で指定したメモリブロック中で動作しているプログラムが存在している状態で実行してはならない。

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

関連項目——_dos_malloc，_dos_mfree，_dos_s_malloc，_dos_s_process

S

_dos_s_process

用　途――サブのメモリ管理を設定する。

書　式――#include <sys/dos.h>

```
int _dos_s_process (int id, int start, int size, int i_len);
```

解　説――_dos_s_process関数はサブのメモリ管理を設定する。*id*にはスレッドIDを指定し，*start*にはサブのメモリ管理の先頭アドレス，*size*にはその領域のバイト数を指定する。

*id*で指定されたスレッドのメモリ管理はこれ以降この範囲内を対象とする。なお，*i_len*には新しいメモリ管理の先頭のメモリブロック（必ず確保される）のバイト数を指定する。

_dos_s_process関数はDOSコール0xFF7Fを発行することによって処理される。

戻り値――正常に終了した場合はメモリブロックの先頭アドレスを返し，失敗した場合は負の値を返す。ただし戻り値が−14ならば，*size*が*i_len*+16よりも小さいことを示す。

注　意――*start*で指定したアドレスから先頭16バイトはメモリ管理領域として使用され破壊される。また，スレッドIDの0番はメインスレッドなので，_dos_s_process関数で変更することはできない。

規　格――*XC*

関連項目――_dos_malloc，_dos_mfree，_dos_s_malloc，_dos_s_mfree

_dos_seek

用　途――ファイルポインタを移動する。

書　式――#include <sys/dos.h>

```
int _dos_seek (int fildes, int offset, int mode);
```

解　説――_dos_seek 関数は *fildes* で指定したファイルのファイルポインタを，*mode* で指定した位置から *offset* バイト移動する。

mode に指定できる値は次のとおり。

- 0　ファイルの先頭
- 1　ファイルの現在位置
- 2　ファイルの終わり

_dos_seek 関数は DOS コール 0xFF42 を発行することによって処理される。

戻り値――成功した場合は移動後のファイルポインタのファイル先頭からのオフセットを返し，失敗した場合は負の値を返す。

規　格――*XC*

S

_dos_send_pr

用　途――指定したスレッドに対してコマンドやデータを送り，スレッドがSLEEPしていたらスレッドを起こす。

書　式――

```
#include <sys/dos.h>
int _dos_send_pr (int my_id, int your_id, int command,
                  char *buff, long len);
```

解　説――_dos_send_pr関数は指定したスレッドに対してコマンドやデータを送り，スレッドがSLEEPしていたらスレッドを起こす。

*my_id*には自分のスレッドIDを指定し，*your_id*には通信したい相手のスレッドIDを指定する。*command*には通信の内容を示す値を設定するが，システム定義のコマンド以外は通信を行うスレッド間で任意に定義することができる。システムで予約しているコマンドは0xFF??であり，現在は次に示すコマンドが定義されている。

- 0xFFF9　スレッドの削除
- 0xFFFB　強制SLEEP状態から起こす(タスク間通信バッファは変化しない)
- 0xFFFC　SLEEPを要求する。起こされたとき，このコマンドが送られていたならば，すぐにタスク間通信バッファのメンバyour_idを−1にしてSLEEPに入るべきである。SLEEPしないでタスク間通信バッファを監視している場合に有効
- 0xFFFF　処理が終わったかどうか調べるためのコマンド。−28が返れば，まだ処理中であることを示す

タスク間通信用のバッファは_prcctrl構造体で定義される。この_prcctrl構造体は次のとおり。

```
struct _prcctrl {
    long length;             /* データバッファの長さ */
    unsigned char *buf_ptr; /* データバッファへのポインタ */
    unsigned short command; /* コマンドバッファ */
    unsigned short your_id; /* 相手のIDのバッファ (-1で通信許可) */
};
```

*your_id*で指定したスレッドの，タスク間通信バッファのメンバyour_idが−1の場合に書き込みが可能である。*my_id*とコマンドをそれぞれ通信バッファのyour_id，commandに設定し，*buff*からのデータを*len*バイトだけ，通信バッファのbuf_ptrで指定した領域へ書き込む。そして，*len*を通信バッファのlengthに設定する。

指定したスレッドがSLEEPしていた場合はスレッドを起こす。特に，0xFFFBのコマンドは特殊処理され，指定したスレッドを起こすだけでタスク間通信バッファの *your_id* が−1でなくてもよく，その他のバッファも変化しない。

len がタスク間通信バッファの length よりも大きい場合や書き込み不可能な場合はエラーとなる。

_dos_send_pr 関数はDOSコール 0xFFFD を発行することによって処理される。

戻り値――失敗した場合は負の値を返す。*my_id* が不正の場合は 0xFFFF00??が返るが，この“0x??”の部分は指定できる *my_id* の最大値を示している。また，*len* が不正の場合は0x80??????を返すが，この“0x??????”の部分は指定できる *len* の最大値を示す。なお，書き込みエラーの場合は−28を返す。

規　格――*XC*

関連項目――_dos_change_pr，_dos_get_pr，_dos_kill_pr，_dos_open_pr，_dos_sleep_pr，_dos_suspend_pr，_dos_time_pr

S

_dos_setblock

用　　途——メモリブロックのサイズを変更する。

書　　式——#include <sys/dos.h>

```
int _dos_setblock (void *memptr, int newsize);
```

解　　説——_dos_setblock 関数は，*memptr* で指定したメモリブロックのサイズを *newsize* に変更する。

memptr には _dos_malloc 関数および _dos_malloc2 関数によって確保した領域へのポインタを指定する。したがって，_dos_malloc 関数および _dos_malloc2 関数によって確保した領域以外のポインタを *memptr* に設定するとエラーとなる。

newsize には，現在のメモリブロックのサイズより大きい値でも小さい値でも指定できる。しかし，メモリブロック内部の整合性は自分で管理する必要がある。

_dos_setblock 関数は DOS コール 0xFF4A を発行することによって処理される。

戻 り 値——正常にメモリサイズを変更できた場合は終了コードを返し，失敗した場合は負の値を返す。ただし，変更できなかった場合は 0x81000000+最大バイト数を返し，完全に変更できるメモリが不足している場合は 0x8200000?を返す。

注　　意——*newsize* に 0x1000000 以上の値を指定した場合は，0xFFFFFF が指定されたとみなす。

規　　格——*XC*

関連項目——_dos_malloc，_dos_malloc2

_dos_setdate

用　　途——現在の日付を設定する。

書　　式——#include <sys/dos.h>

```
int _dos_setdate (int date);
```

解　　説——_dos_setdate 関数は現在の日付を設定する。

date に指定するビットフィールドの内容は次のとおり。

00000000 00000000 yyyyyyym mmmddddd

- ddddd　　日 (1 ～ 31)
- mmmm　　月 (1 ～ 12)
- yyyyyyy　　年 (0 ～ 99) (1980 年からの相対年数)

_dos_setdate 関数は DOS コール 0xFF2B を発行することによって処理される。

戻 り 値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。

規　　格——*XC*

関連項目——_dos_getdate，_dos_gettim2，_dos_gettime，_dos_settim2，_dos_settime

S

_dos_setenv

用　　途——環境変数を設定する。

書　　式——#include <sys/dos.h>

```
int _dos_setenv (const char *name,
                 const char *env, const char *buff);
```

解　　説——_dos_setenv 関数は *env* で指定した環境から *name* で指定した環境変数を検索し，その値を *buff* の指す領域に格納する。

env に 0 を設定すると，親プロセスの環境を対象とする。また，*buff* に 0 を設定すると *name* の環境変数を削除する。

_dos_setenv 関数は DOS コール 0xFF52(0xFF82) を発行することによって処理される。

戻 り 値——失敗した場合は負の値を返す。

注　　意——環境変数には 255 バイトまでしか設定できない。

互 換 性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格——*XC*

関連項目——_dos_getenv

_dos_setpdb

用　　途——管理プロセスを移す。

書　　式——
```
#include <sys/dos.h>
void _dos_setpdb (const struct _psp *pdbaddr);
```

解　　説——_dos_setpdb 関数は *pdbaddr* で指定したプロセスへ実行制御を移す。

pdbaddr は制御を移すプログラムの先頭アドレス−0xF0 のアドレスであり，_dos_getpdb 関数の戻り値でなくてはならない。

_dos_setpdb 関数は DOS コール 0xFF50(0xFF80) を発行することによって処理される。

戻 り 値——制御を移す前のプロセス管理ポインタを返す。

互 換 性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　　格——*XC*

関連項目——_dos_getpdb

S

_dos_settim2

用　　途——現在の時刻を設定する。

書　　式——#include <sys/dos.h>

```
int _dos_settim2 (int time);
```

解　　説——_dos_settim2 関数は現在の時刻を *time* に設定する。

time に指定するビットフィールドの内容は次のとおり。

00000000 000hhhhh 00mmmmmm 00ssssss

- ssssss　　秒 (0 ～ 59)
- mmmmmm　　分 (0 ～ 59)
- hhhhh　　時 (0 ～ 23)

_dos_settim2 関数は DOS コール 0xFF28 を発行することによって処理される。

戻 り 値——成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。

規　　格——*XC*

関連項目——_dos_getdate，_dos_gettim2，_dos_gettime，_dos_setdate，_dos_settime

_dos_settime

用　途──現在の時刻を設定する。

書　式──
```
#include <sys/dos.h>
int _dos_settime (int time);
```

解　説──_dos_settime 関数は現在の時刻を *time* に設定する。

time に指定するビットフィールドの内容は次のとおり。

00000000 00000000 hhhhhmmm mmmsssss

- sssss　秒 (0 ～ 29) (実際の値は 2 を乗じる)
- mmmmmm　分 (0 ～ 59)
- hhhhh　時 (0 ～ 23)

_dos_settime 関数は DOS コール 0xFF2D を発行することによって処理される。

戻り値──成功した場合は 0 を含む正の値を返し，失敗した場合は負の値を返す。

規　格──*XC*

関連項目──_dos_getdate，_dos_gettim2，_dos_gettime，_dos_setdate，_dos_settim2

S

_dos_sleep_pr

用　途——カレントスレッドSLEEP状態にする。

書　式——#include <sys/dos.h>

```
long _dos_sleep_pr (long time);
```

解　説——_dos_sleep_pr関数はカレントスレッドをSLEEP状態にする。*time*には待ち時間をミリ秒単位で指定するが，0を指定した場合は永久にSLEEPする。

SLEEP状態に入ったスレッド_dos_send_pr関数によって強制的に起こすことができる。_dos_send_pr関数のコマンドが0xFFFBの場合，タスク間通信バッファは変化しないが，それ以外のコマンドの場合はタスク間通信バッファは次のようになる。

```
struct _prcctrl {
    long length;              /* データバッファの長さ */
    unsigned char *buf_ptr; /* データバッファのポインタ（変化しない）*/
    unsigned short command; /* コマンド番号 */
    unsigned short your_id; /* 起こしてくれた相手のID */
};
```

SLEEPする前に，データバッファの内容を処理してからデータバッファのアドレスとバイト数を設定し，タスク間通信バッファのメンバyour_idに−1を設定すること。そうすることで，他のスレッドからの通信を許可することになる。

SLEEPしていない場合でも，_dos_send_pr関数によってデータが送られてくる場合がある。その場合はSLEEPしてもすぐに起こされ，設定した待ち時間が戻り値として返される。

_dos_sleep_pr関数はDOSコール0xFFFCを発行することによって処理される。

戻り値——待ち時間が経過して起きた場合は−1を返し，タスク間通信バッファは変化しない。

待ち時間が経過したが，_dos_suspend_pr関数で止められていたため_dos_send_pr関数によって起こされた場合は−2を返す。タスク間通信バッファは_dos_send_pr関数により変化する。

戻り値が−1でも−2でもない場合は，待ち時間の残り時間をミリ秒単位で返す。タスク間通信バッファは，_dos_send_pr関数により変化する。

規　格——*XC*

関連項目——_dos_change_pr，_dos_get_pr，_dos_kill_pr，_dos_open_pr，_dos_send_pr，_dos_suspend_pr，_dos_time_pr，

_dos_super

用　途——スーパーバイザモードとユーザモードとを切り替える。

書　式——#include <sys/dos.h>

```
int _dos_super (int stack);
```

解　説——_dos_super 関数はスーパーバイザモードとユーザモードの切り替えを行う。

stack が 0 のときには USP レジスタの値を SSP レジスタにセットして，スーパーバイザモードへ切り替える。*stack* が 0 以外のときには *stack* を SSP レジスタにセットして，ユーザモードへ切り替える。

_dos_super 関数は DOS コール 0xFF20 を発行することによって処理される。

戻り値——成功した場合，*stack* が 0 のときは切り替える前の SSP レジスタの値を返し，失敗した場合は負の値を返す。

規　格——*XC*

S

_dos_super_jsr

用　　途——スーパーバイザ領域のプログラムをサブルーチンコールする。

書　　式——

```
#include <sys/dos.h>
void _dos_super_jsr (void (*func) (), const struct _dregs *inregs,
                     struct _dregs *outregs);
```

解　　説——_dos_super_jsr 関数はスーパーバイザ領域のプログラムをサブルーチンコールする。

*func*にはサブルーチンコールしたいプログラムのエントリアドレスを指定し，*inregs*には呼び出したプログラムへ渡すレジスタを設定する。サブルーチンコールの後，*outregs*にはサブルーチンからリターンするときのレジスタの値が設定される。

_dos_super_jsr 関数は DOS コール 0xFFF6 を発行することによって処理される。

戻 り 値——なし。

注　　意——sr レジスタの値は変化しない。また，usp/ssp レジスタはどのような値がプログラムに渡されるか不定なので，スタックを使った引数渡しは行えない。

サブルーチンコールした結果，バスエラーやアドレスエラーによって実行が失敗した場合でもエラーに対する処理は行わない。

この DOS コールのエントリを，_dos_intvcs 関数によって変更してはならない。

規　　格——*XC*

_dos_suspend_pr

用　　途——スレッドを強制的に SLEEP 状態にする。

書　　式——
```
#include <sys/dos.h>
int _dos_suspend_pr (int id);
```

解　　説——_dos_suspend_pr 関数は *id* で指定したスレッドを強制的に SLEEP 状態にする。

SLEEP 状態になったスレッドは，_dos_send_pr 関数によって起こされるまで SLEEP する。

_dos_suspend_pr 関数は DOS コール 0xFFFB を発行することによって処理される。

戻 り 値——正常終了した場合は 0 を返し，失敗した場合は負の値を返す。*id* が不正の場合は 0xFFFF00??を返すが，この "0x??" の部分が指定できる *id* の最大値を示している。また，自分自身を指定した場合は−1 を返す。

規　　格——*XC*

関連項目——_dos_change_pr，_dos_get_pr，_dos_kill_pr，_dos_open_pr，_dos_send_pr，_dos_sleep_pr，_dos_time_pr

S

_dos_symlink

用　　途――シンボリックリンクファイルを作成する。

書　　式――#include <sys/dos.h>

```
int _dos_symlink (const char *src, const char *dst);
```

解　　説――_dos_symlink 関数は，*src*をリンク先とするシンボリックリンクファイル *dst*を作成する。ただし *dst*と同名のファイルがすでに存在する場合は失敗する。

_dos_symlink 関数は **Human68k** の DOS コールではない。

戻 り 値――成功した場合は 0 を返し，失敗した場合は負の値を返す。

注　　意――_dos_symlink 関数の動作はすべて **lndrv** に依存するため，**lndrv** 以外のシンボリックリンクドライバでは使用することができない。また，将来 **lndrv** が変更されたりしても動作しなくなる可能性がある。

規　　格――***Project LIBC Group***

関連項目――_dos_importlnenv，_dos_lfiles，_dos_readlink

_dos_time_pr

用　途──現在のタイマのカウンタ値を返す。

書　式──#include <sys/dos.h>

```
int _dos_time_pr (void);
```

解　説──_dos_time_pr 関数は現在のタイマのカウンタ値 (ミリ秒単位) を返す。

_dos_time_pr 関数は DOS コール 0xFFFE を発行することによって処理される。

戻り値──戻り値のビットフィールドの内容は次のとおり。時/分/秒はすべて BCD 2 桁である。

```
00000000 HHHHHHHH MMMMMMMM SSSSSSSS
```

- SSSSSSSS　秒 (0 ～ 59)
- MMMMMMMM　分 (0 ～ 59)
- HHHHHHHH　時 (0 ～ 23)

規　格──*XC*

関連項目──_dos_change_pr, _dos_get_pr, _dos_kill_pr, _dos_open_pr, _dos_send_pr, _dos_sleep_pr, _dos_suspend_pr

T

_dos_unlink

用　途——ハードリンクファイルを削除する。

書　式——#include <sys/dos.h>

```
int _dos_unlink (const char *linkpath);
```

解　説——_dos_unlink 関数は，*linkpath* で指定したハードリンクファイルを削除する。

_dos_unlink 関数は **Human68k** の DOS コールではない。

戻り値——成功した場合は 0 を返し，失敗した場合は負の値を返す。

注　意——**Human68k** にはハードリンクの概念がないため，_dos_unlink 関数では実現するために非常に危険な方法を採用している。もしハードリンク以外のファイルを削除すると「どこからも指されていないセクタ」を作成してしまうことになる。不用意な使用はファイルシステムの破壊になりかねないので注意すること。

規　格——***Project LIBC Group***

関連項目——_dos_link

_dos_verify

用　　途——ベリファイフラグを設定する。

書　　式——#include <sys/dos.h>

```
void _dos_verify (int flag);
```

解　　説——_dos_verify 関数はベリファイフラグを設定する。

*flag*に指定できる値は次のとおり。

- 0　ベリファイしない
- 1　ベリファイする

_dos_verify 関数は DOS コール 0xFF2E を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

関連項目——_dos_verifyg

_dos_verifyg

用　途——ベリファイフラグの設定状況を取得する。

書　式——
```
#include <sys/dos.h>
int _dos_verifyg (void);
```

解　説——_dos_verifyg 関数はベリファイフラグの設定状況を取得する。

_dos_verifyg 関数は DOS コール 0xFF54(0xFF84) を発行することによって処理される。

戻り値——ベリファイフラグの設定状況を返す。戻り値は次のとおり。

- 0　ベリファイしない
- 1　ベリファイする

互換性——**Human68k** ver.3 では DOS コールの番号が変更 (+0x30) されたことに注意すること。

規　格——*XC*

関連項目——_dos_verify

_dos_vernum

用　途——**Human68k** のバージョン番号を返す。

書　式——

```
#include <sys/dos.h>
int __const _dos_vernum (void);
```

解　説——_dos_vernum 関数は **Human68k** のバージョン番号を返す。

戻り値のバージョン番号の値は次に示す形式をとる。

- 下位 16 ビット　　'68'
- 上位 16 ビット　　バージョン番号の整数部× 256+小数部

_dos_vernum 関数は DOS コール 0xFF30 を発行することによって処理される。

戻り値——**Human68k** のバージョン番号を返す。

規　格——*XC*

_dos_wait

用　途——自プロセスが直前に実行した子プロセスの終了コードを返す。

書　式——#include <sys/dos.h>

```
int _dos_wait (void);
```

解　説——_dos_wait 関数は自プロセスが直前に実行した子プロセスの終了コードを取得する。

_dos_wait 関数は DOS コール 0xFF4D を発行することによって処理される。

戻り値——子プロセスの終了コードを返す。

規　格——*XC*

_dos_write

用　途——ファイルへデータを書き込む。

書　式——#include <sys/dos.h>

```
int _dos_write (int fildes, const char *buff, int size);
```

解　説——_dos_write 関数は，*buff* が指す領域から *size* バイト分のデータを *fildes* で指定したファイルへ書き出す。

_dos_write 関数は DOS コール 0xFF40 を発行することによって処理される。

戻り値——成功した場合は実際に書き込んだバイト数を返し，失敗した場合は負の値を返す。ただし，*size* よりも戻り値が小さい場合や 0 が返った場合は，ディスクフル (空き容量なし) が発生したことを示す。

注　意——ディスクフルではエラー (戻り値が負の値) とならず，0 が返されることに注意すること。

規　格——*XC*

関連項目——_dos_read

W

Chapter 3

IOCS コールライブラリ

本章には X68000 の IOCS コールライブラリのマニュアルを掲載します。*LIBC* では従来の *XC* との互換性を考慮し，名前が異なる以外は基本的な動作はすべて同じに作成されています。

本書では IOCS コールについては関数ごとのマニュアルにとどめ，各デバイス (LSI) などの詳しい説明はしません。詳しく知りたい場合は，他の書籍などを参照してください。

_iocs_abortjob

用　途——アボート処理を行う。

書　式——
```
#include <sys/iocs.h>
void _iocs_abortjob (void);
```

解　説——_iocs_abortjob 関数はアボート処理を行う。この機能は **Human68k** が使用するのでユーザは使用しないこと。

_iocs_abortjob 関数は IOCS コール 0xFF を発行することによって処理される。

戻り値——リターンしないで，エラーアボートルーチン (OS のエラーが出たとき，中止<A>を選ぶ) へ制御を移す。

規　格——*XC*

_iocs_abortrst

用　途――アボートするために環境の再設定を行う。

書　式――#include <sys/iocs.h>

```
void _iocs_abortrst (void);
```

解　説――_iocs_abortrst 関数はプログラムの処理をアボートするためのフラグをリセットし，環境を再設定する。この機能は **Human68k** が使用するので，ユーザは使用しないこと。

_iocs_abortrst 関数は IOCS コール 0xFD を発行することによって処理される。

戻り値――なし。

規　格――*XC*

_iocs_adpcmain

用　途――ADPCMからアレイチェインモードでデータを入力する。

書　式――#include <sys/iocs.h>

```
int _iocs_adpcmain (const struct _chain *tbl, int mode, int cnt);
```

解　説――_iocs_adpcmain関数はADPCMからアレイチェインモードでデータを入力する。*tbl*に指定する_chain構造体は次のとおり。

```
struct _chain {
    void *addr;         /* データバッファの先頭アドレス */
    unsigned short len; /* データバッファの長さ */
};
```

*mode*については_iocs_adpcminp関数を参照のこと。また，*cnt*にはチェインテーブルの個数を指定する。

_iocs_adpcmain関数はIOCSコール0x63を発行することによって処理される。

戻り値――なし。

規　格――*XC*

関連項目――_iocs_adpcminp

_iocs_adpcmaot

用　途——ADPCM へアレイチェインモードでデータを出力する。

書　式——#include <sys/iocs.h>

```
int _iocs_adpcmaot (const struct _chain *tbl, int mode, int cnt);
```

解　説——_iocs_adpcmaot 関数は ADPCM へアレイチェインモードでデータを出力する。*tbl* に指定する _chain 構造体は次のとおり。

```
struct _chain {
    void *addr;          /* データの先頭アドレス */
    unsigned short len; /* データの長さ */
};
```

mode については _iocs_adpcmout 関数を参照のこと。また，*cnt* にはチェインテーブルの個数を指定する。

_iocs_adpcmaot 関数は IOCS コール 0x62 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_iocs_adpcmout

_iocs_adpcminp

用　　途——ADPCMからデータを入力する。

書　　式——#include <sys/iocs.h>

```
int _iocs_adpcminp (void *addr, int mode, int len);
```

解　　説——_iocs_adpcminp 関数はADPCMからデータを入力する。

*addr*には入力データの格納アドレスを指定し，*len*には入力データ長を指定する。この指定が 0xFF00 バイト以上の場合，入力処理を 0xFF00 バイト単位に行うので，関数の処理が完了するには時間がかかる。

*mode*にはサンプリング周波数×256+出力モードを指定する。

サンプリング周波数の指定は次のとおり。

- 0　3.9KHz(1950バイト/秒)
- 1　5.2KHz(2600バイト/秒)
- 2　7.8KHz(3900バイト/秒)
- 3　10.4KHz(5200バイト/秒)
- 4　15.6KHz(7800バイト/秒)

出力モードはモニタ出力の出力モードであり，出力モードの指定は次のとおり。

- 0　音声出力カット
- 1　音声出力左
- 2　音声出力右
- 3　音声出力両方

_iocs_adpcminp 関数はIOCSコール 0x61 を発行することによって処理される。

戻り値——なし。

注　　意——データの入力自体はモノラルで行われる。

規　　格——*XC*

_iocs_adpcmlin

用　途——ADPCM からリンクアレイチェインモードでデータを入力する。

書　式——#include <sys/iocs.h>

```
int _iocs_adpcmlin (const struct _chain2 *tbl, int mode);
```

解　説——_iocs_adpcmlin 関数は ADPCM からリンクアレイチェインモードでデータを入力する。*tbl*に指定する_chain2 構造体は次のとおり。

```
struct _chain2 {
    void *addr;                 /* データバッファの先頭アドレス */
    unsigned short len;         /* データバッファの長さ */
    const struct _chain2 *next; /* 次のテーブルアドレス */
};
```

最後のテーブルのメンバnext には 0 を指定する。また，*mode*については_iocs_adpcminp 関数を参照のこと。

_iocs_adpcmlin 関数は IOCS コール 0x65 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_iocs_adpcminp

_iocs_adpcmlot

用　途——ADPCMへリンクアレイチェインモードでデータを出力する。

書　式——#include <sys/iocs.h>

```
int _iocs_adpcmlot (const struct _chain2 *tbl, int mode);
```

解　説——_iocs_adpcmlot関数はADPCMへリンクアレイチェインモードでデータを出力する。*tbl*に指定する_chain2構造体は次のとおり。

```
struct _chain2 {
    void *addr;                 /* データの先頭アドレス */
    unsigned short len;         /* データの長さ */
    const struct _chain2 *next; /* 次のテーブルアドレス */
};
```

最後のテーブルのメンバnextには0を指定する。また，*mode*については_iocs_adpcmout関数を参照のこと。

_iocs_adpcmlot関数はIOCSコール0x64を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_iocs_adpcmout

_iocs_adpcmmod

用　途——ADPCM の実行を制御する。

書　式——

```
#include <sys/iocs.h>
void _iocs_adpcmmod (int mode);
```

解　説——_iocs_adpcmmod 関数は ADPCM の実行を制御する。

mode に指定できる値は次のとおり。

- 0　終了
- 1　中止
- 2　再開

_iocs_adpcmmod 関数は IOCS コール 0x67 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_adpcmout

用　途——ADPCMへデータを出力する。

書　式——#include <sys/iocs.h>

```
int _iocs_adpcmout (const void *addr, int mode, int len);
```

解　説——_iocs_adpcmout関数はADPCMへデータを出力する。

*addr*には出力データの格納アドレスを指定し，*len*には出力データ長を指定する。ただし，この指定が0xFF00バイト以上の場合，出力処理を0xFF00バイト単位に行うので，関数の処理が完了するには時間がかかる。

*mode*にはサンプリング周波数×256+出力モードを指定する。サンプリング周波数の指定は次のとおり。

- 0　3.9KHz(1950バイト/秒)
- 1　5.2KHz(2600バイト/秒)
- 2　7.8KHz(3900バイト/秒)
- 3　10.4KHz(5200バイト/秒)
- 4　15.6KHz(7800バイト/秒)

出力モードの指定は次のとおり。

- 0　音声出力カット
- 1　音声出力左
- 2　音声出力右
- 3　音声出力両方

_iocs_adpcmout関数はIOCSコール0x60を発行することによって処理される。

戻り値——なし。

規　格——*XC*

A

_iocs_adpcmsns

用　途——ADPCM の実行モードを調べる。

書　式——#include <sys/iocs.h>

```
int _iocs_adpcmsns (void);
```

解　説——_iocs_adpcmsns 関数は ADPCM の実行モードを調べる。

_iocs_adpcmsns 関数は IOCS コール 0x66 を発行することによって処理される。

戻り値——ビットフィールドで次の値を返す。

- 0x00　何もしていない
- 0x02　出力中 (_iocs_adpcmout 関数実行中)
- 0x04　入力中 (_iocs_adpcminp 関数実行中)
- 0x12　出力中 (_iocs_adpcmaot 関数実行中)
- 0x14　入力中 (_iocs_adpcmain 関数実行中)
- 0x22　出力中 (_iocs_adpcmlot 関数実行中)
- 0x24　入力中 (_iocs_adpcmlin 関数実行中)

規　格——*XC*

関連項目——_iocs_b_adpcmain, _iocs_b_adpcmaot, _iocs_b_adpcminp, _iocs_b_adpcmlin, _iocs_b_adpcmlot, _iocs_b_adpcmout

_iocs_akconv

用　途——ANKコードからシフトJIS漢字コードに変換する。

書　式——

```
#include <sys/iocs.h>
int _iocs_akconv (int mode, int code);
```

解　説——_iocs_akconv関数はANKコードからシフトJIS漢字コードへの変換を行う。*code*には変換するANKコード(0x20 ~ 0x7E, 0xA1 ~ 0xDF)を指定する。

また，*mode*には文字種を指定する。文字種は次のとおり。

- 0　ひらがな
- 1　カタカナ

_iocs_akconv関数はIOCSコール0xA2を発行することによって処理される。

戻り値——変換したシフトJIS漢字コードを返すが，上位16ビットが0xFFFFならばエラーが発生したことを示す。

規　格——*XC*

_iocs_alarmget

用　途——アラームの時間と処理アドレスを読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_alarmget (int *datetime, int *offtime, int *mode);
```

解　説——_iocs_alarmget 関数はアラームの時間と処理アドレスを読み込む。

_iocs_alarmget 関数は IOCS コール 0x5F を発行することによって処理される。

戻り値——*datetime*, *offtime*, *mode* で指定した領域にアラームの設定状況を格納する。それぞれの内容については, _iocs_alarmset 関数を参照のこと。

規　格——*XC*

関連項目——_iocs_alarmset

_iocs_alarmmod

用　途――アラームの禁止/許可を設定する。

書　式――#include <sys/iocs.h>

```
int _iocs_alarmmod (int mode);
```

解　説――_iocs_alarmmod 関数はアラームの禁止/許可および現在の状態の通知を行う。

mode には，次の値を設定する。

- 0　禁止
- 1　許可
- 2　通知要求

_iocs_alarmmod 関数はIOCS コール 0x5D を発行することによって処理される。

戻り値――設定されたアラームの状態を返す。返される値は次のとおり。

- 0　禁止
- 1　許可

規　格――*XC*

_iocs_alarmset

用　途——アラームの時間と処理アドレスの設定を行う。

書　式——#include <sys/iocs.h>

```
int _iocs_alarmset (int datetime, int offtime, int mode);
```

解　説——_iocs_alarmset 関数はアラームの時間と処理アドレスを設定する。

datetime に指定するビットフィールドの内容は次のとおり。日/時/分はいずれもBCD2桁である。曜日に 0x0F，日/時/分に 0xFF を指定すると，それぞれ無指定となる。ただし，すべてを無指定にはしないこと。

```
0000WWWW DDDDDDDD HHHHHHHH MMMMMMMM
```

- WWWW　　曜日 (0x00 ～ 0x06) (0x00 ＝日曜日)
- DDDDDDDD　　日 (0x01 ～ 0x31)
- HHHHHHHH　　時 (0x00 ～ 0x23)
- MMMMMMMM　　分 (0x00 ～ 0x59)

offtime にはコンピュータがオフされるまでの時間を指定する。*offtime* の値は次のとおり。

- 0　　いつまでも OFF しない
- 0 以外　　OFF するまでの時間 (1 分単位)

mode には処理アドレスを指定する。*mode* の値は次のとおり。0x01 ～ 0x3F のテレビコントロールについては_iocs_tvctrl 関数を参照のこと。また，処理アドレスで指定されるプログラムの先頭は 0x60(bra 命令) でなければならない。

- 0　　パワー ON，テレビオン&コンピュータモード
- −1　　パワー ON のみ，テレビコントロールはなし
- 1 ～ 0x3F　　テレビコントロール
- 0x40 ～ 0xFFFFFFFE　　処理アドレス

_iocs_alarmset 関数は IOCS コール 0x5E を発行することによって処理される。

戻り値——失敗した場合は−1 を返す。

規　格——*XC*

_iocs_apage

用　　途――グラフィック画面の書き込みページを設定する。

書　　式――#include <sys/iocs.h>

```
int _iocs_apage (int mode);
```

解　　説――_iocs_apage 関数はグラフィック画面の書き込みページの設定を行う。

*mode*は書き込みページを指定する。指定する値の内容は次のとおり。

- 0　　　書き込みページに0ページを指定
- 1　　　書き込みページに1ページを指定
- 2　　　書き込みページに2ページを指定
- 3　　　書き込みページに3ページを指定
- −1　　現在の書き込みページを調べる

_iocs_apage 関数はIOCSコール 0xB1 を発行することによって処理される。

戻り値――成功した場合，*mode*が−1のときは0〜3を返し，それ以外の場合は0を返す。

また，失敗した場合は次の値を返す。

- −1　　グラフィックは使用できない
- −2　　規定外のページを指定した
- −3　　指定したページは現在の画面モードでは設定できない

規　　格――*XC*

_iocs_b_badfmt

用　途——不良トラックを登録する。

書　式——#include <sys/iocs.h>

```
int _iocs_b_badfmt (int drive, int recno, int mode);
```

解　説——_iocs_b_badfmt関数は，ハードディスクの不良トラックを使用不可トラックとして登録する。

*drive*には*pda*×256を指定する。*pda*については_iocs_b_seek関数を参照のこと。

*recno*には256バイト単位のレコード番号を指定し，*mode*にはインタリーブコードを指定する。通常，10Mバイトドライブの場合は6を，20/40Mバイトドライブの場合は1を指定する。

_iocs_b_badfmt関数はIOCSコール0x4Bを発行することによって処理される。

戻り値——成功した場合は0を含む正の値を返し，失敗した場合は0xFFFFFF??を返す。

規　格——*XC*

関連項目——_iocs_b_seek

_iocs_b_bpeek

用　　途――メモリから1バイトデータ読み込む。

書　　式――

```
#include <sys/iocs.h>
int _iocs_b_bpeek (const void *addr);
```

解　　説――_iocs_b_bpeek 関数は，*addr* で指定したアドレスから1バイトのデータを読み込む。主にユーザモードでスーパーバイザエリアをアクセスするために使用する。

_iocs_b_bpeek 関数はIOCSコール0x82を発行することによって処理される。

戻 り 値――読み込んだバイトデータを返す。

規　　格――*XC*

_iocs_b_bpoke

用　途——メモリに1バイトデータ書き込む。

書　式——

```
#include <sys/iocs.h>
void _iocs_b_bpoke (void *addr, int data);
```

解　説——_iocs_b_bpoke 関数は *addr* で指定したアドレスに，*data* で指定したデータを1バイト書き込む。主に，ユーザモードでスーパーバイザエリアをアクセスするために使用する。

_iocs_b_bpoke 関数は IOCS コール 0x86 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_clr

用　途——カーソル位置を基準としてテキスト画面をクリアする。

書　式——

```
#include <sys/iocs.h>
void _iocs_b_clr_ed (void);
void _iocs_b_clr_st (void);
void _iocs_b_clr_al (void);
```

解　説——_iocs_b_clr_ed 関数は，現在のカーソル位置から最終行右端までのテキスト画面をクリアする。

_iocs_b_clr_st 関数は，先頭行左端から現在のカーソル位置までのテキスト画面をクリアする。

_iocs_b_clr_al 関数は，テキスト画面全体をクリアする。カーソルはホームポジションへ戻る。

これらの関数は IOCS コール 0x2A を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_color

用　途——表示属性を設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_b_color (int color);
```

解　説——_iocs_b_color 関数は，画面の表示属性を *color* で指定した属性に設定する。

color に指定できる値は次のとおり。

- 0　黒
- 1　水色
- 2　黄色
- 3　白
- 4　黒
- 5　水色の強調
- 6　黄色の強調
- 7　白の強調
- 8　黒
- 9　水色のリバース
- 10　黄色のリバース
- 11　白のリバース
- 12　黒
- 13　水色の強調，リバース
- 14　黄色の強調，リバース
- 15　白の強調，リバース

_iocs_b_print 関数は IOCS コール 0x22 を発行することによって処理される。

戻り値——成功した場合は設定前の属性を返し，失敗した場合は−1 を返す。

規　格——*XC*

_iocs_b_consol

用　　途――テキストの表示範囲を設定する。

書　　式――#include <sys/iocs.h>

```
int _iocs_b_consol (int xs, int ys, int xl, int yl);
```

解　　説――_iocs_b_consol 関数は表示範囲の設定を行う。

*xs*には先頭の X 座標 (0 ～ 1008 までの 16 の倍数) を指定し，*ys*には先頭の Y 座標 (0 ～ 1020 までの 4 の倍数) を指定する。*xl*には X の桁数−1 (0 ～ 127 で 8 ドット単位) を指定し，*yl*には Y の桁数−1 (0 ～ 63 で 16 ドット単位) を指定する。

ただし *xs* と *ys* の両方が−1 であるか，*xl* と *yl* の両方が−1 だった場合は，それぞれ直前までの値を引き継ぐ。

_iocs_b_consol 関数は IOCS コール 0x2E を発行することによって処理される。

戻 り 値――上位 16 ビットに直前の X の桁数−1 を返し，下位 16 ビットに直前の Y の桁数−1 を返す。

規　　格――*XC*

_iocs_b_curoff

用　　途——カーソルを消去する。

書　　式——`#include <sys/iocs.h>`

```
void _iocs_b_curoff (void);
```

解　　説——`_iocs_b_curoff` 関数はカーソルを消去する。

`_iocs_b_curoff` 関数は IOCS コール `0x1F` を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

関連項目——`_iocs_b_curon`

_iocs_b_curon

用　　途——カーソルを表示する。

書　　式——#include <sys/iocs.h>

```
void _iocs_b_curon (void);
```

解　　説——_iocs_b_curon 関数はカーソルを表示する。

_iocs_b_curon 関数は IOCS コール 0x1E を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

関連項目——_iocs_b_curoff

_iocs_b_del

用　途——カーソル表示行を削除する。

書　式——#include <sys/iocs.h>

```
void _iocs_b_del (int cnt);
```

解　説——_iocs_b_del 関数は，現在のカーソル行から *cnt* 行を削除する。ただし，*cnt* に 0 を指定すると 1 とみなされる。また，カーソル位置の X 座標は 0 となる。

_iocs_b_del 関数は IOCS コール 0x2D を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_down

用　途――カーソル位置を指定行数だけ下へ移動する。

書　式――

```
#include <sys/iocs.h>
void _iocs_b_down (int cnt);
```

解　説――_iocs_b_down 関数はカーソル位置を *cnt* 行下へ移動する。カーソル位置が下端を越える場合でも，スクロールアップは行われない。なお，*cnt* に 0 を指定すると 1 とみなされる。

_iocs_b_down 関数は IOCS コール 0x27 を発行することによって処理される。

戻り値――なし。

規　格――*XC*

_iocs_b_down_s

用　途——カーソル位置を１行下へ移動する。

書　式——#include <sys/iocs.h>

```
void _iocs_b_down_s (void);
```

解　説——_iocs_b_down_s 関数はカーソル位置を１行下へ移動する。現在のカーソル位置が最下行だった場合はスクロールアップが行われる。

_iocs_b_down_s 関数は IOCS コール 0x24 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_drvchk

用　途――フロッピーディスクドライブの状態の検査/設定を行う。

書　式――
```
#include <sys/iocs.h>
int _iocs_b_drvchk (int drive, int mode);
```

解　説――_iocs_b_drvchk 関数は *drive* で指定したドライブの状態の検査と設定を行う。

drive には *pda*×256 を指定する。*pda* については _iocs_b_seek 関数を参照のこと。

mode には次の値を指定する。

- 0　状態の検査のみ行う
- 1　イジェクトする(イジェクト禁止の場合は不可)
- 2　イジェクト禁止1(*mode*=1 のイジェクトも禁止)
- 3　イジェクト許可1
- 4　ディスクがセットされていないとき LED 点滅
- 5　ディスクがセットされていないとき LED 消灯
- 6　イジェクト禁止2(*mode*=1 のイジェクトも禁止)
- 7　イジェクト許可2
- 8　前回の検査後にイジェクトが行われたかどうかを調べる

_iocs_b_drvchk 関数は IOCS コール 0x4E を発行することによって処理される。

戻り値――*mode* が 0～7 のとき，次のようなビットフィールドを返す。

- ビット0　メディア誤挿入
- ビット1　メディア挿入
- ビット2　ノットレディ
- ビット3　ライトプロテクト
- ビット4　ユーザによるイジェクト禁止
- ビット5　バッファあり
- ビット6　イジェクト禁止
- ビット7　LED 点滅

ビット2，3は *mode* が0の場合だけ値をもち，それ以外の場合は0となる。また *mode* が8の場合は，1ならばイジェクトしていないことを，−1ならばイジェクトしたことを示す。

注　　意——*mode*の 6 ～ 8 は **Human68k** が使用するのでユーザは使用しないこと。

規　　格——*XC*

関連項目——_iocs_b_seek

_iocs_b_drvsns

用　途——ディスクのステータス情報を調べる。

書　式——#include <sys/iocs.h>

```
int _iocs_b_drvsns (int drive);
```

解　説——_iocs_b_drvsns 関数は *drive* で指定したドライブのステータス情報を調べる。

drive には *pda*×256 を指定する。*pda* については _iocs_b_seek 関数を参照のこと。

_iocs_b_drvsns 関数は IOCS コール 0x44 を発行することによって処理される。

戻り値——成功した場合，ハードディスクならば 0 を含む正の値を，フロッピーディスクならば FDC のステータス情報 (負の値もあり得る) を返す。また，失敗した場合は 0xFFFFFF??を返す。

FDC のステータス情報については，_iocs_b_dskini 関数を参照のこと。

規　格——*XC*

関連項目——_iocs_b_dskini，_iocs_b_seek

_iocs_b_dskini

用　途——ディスクインタフェイスを初期化する。

書　式——**フロッピーディスクの場合**

```
#include <sys/iocs.h>
int _iocs_b_dskini (int drive, const void *data, int offtime);
```

ハードディスクの場合

```
#include <sys/iocs.h>
int _iocs_b_dskini (int drive, const void *data);
```

解　説——_iocs_b_dskini 関数は，*drive* で指定したドライブに対応するインタフェイスを初期化する。

drive には *pda* ×256 を指定する。*pda* については _iocs_b_seek 関数を参照のこと。

フロッピーディスクの場合，*data* には SPECIFY コマンドのデータアドレスを指定するが，0 の場合はデフォルトの値 (0x03，0xD0，0x10) が設定される。また，*offtime* には**モーターオフ**までの時間 (単位：1/100 秒) を設定する。これも 0 の場合はデフォルト値 (200，2 秒) が設定される。

ハードディスクの場合，*data* には**ドライブパラメータ**のデータアドレスを指定するが，0 の場合はデフォルト値 (0x01，0x01，0x00，0x03，0x01，0x35，0x80，0x00，0x00，0x00) が設定される。

_iocs_b_dskini 関数は IOCS コール 0x43 を発行することによって処理される。

戻り値——成功した場合，ハードディスクならば 0 を含む正の値を，フロッピーディスクならば FDC のステータス情報 (負の値もあり得る) を返す。また，失敗した場合は 0xFFFFFF??を返す。

FDC のステータス情報は次のとおり。

- ビット 31 ～ 24　　リザルトステータス 3
- ビット 23 ～ 0　　不定

リザルトステータス 3 の内容は次のとおり。

- ビット 7　　Fault 信号の状態
- ビット 6　　Write Protected 信号の状態
- ビット 5　　Ready 信号の状態
- ビット 4　　Track0 信号の状態
- ビット 3　　Tow Side 信号の状態

- ビット2　Side Select 信号の状態
- ビット1　Unit Select1 信号の状態
- ビット0　Unit Select0 信号の状態

規　格—— *XC*

関連項目—— _iocs_b_seek,　_iocs_b_verify

B

_iocs_b_eject

用　途——ディスクのイジェクトを行う。

書　式——#include <sys/iocs.h>

```
int _iocs_b_eject (int drive);
```

解　説——_iocs_b_eject 関数は，*drive* で指定したドライブのメディアをイジェクトする。

_iocs_b_eject 関数はイジェクト禁止状態でもイジェクトを行う。また，ハードディスクの場合はシッピング (未使用シリンダへヘッドをシーク) する。

drive には *pda* ×256 を指定する。*pda* については _iocs_b_seek 関数を参照のこと。

_iocs_b_eject 関数は IOCS コール 0x4F を発行することによって処理される。

戻り値——成功した場合は 0 を含む正の値を返し，失敗した場合は 0xFFFFFF??を返す。

規　格——*XC*

関連項目——_iocs_b_seek

_iocs_b_era

用　　途——カーソル位置を基準としてテキスト画面をクリアする。

書　　式——

```
#include <sys/iocs.h>
void _iocs_b_era_ed (void);
void _iocs_b_era_st (void);
void _iocs_b_era_al (void);
```

解　　説——_iocs_b_era_ed 関数は，現在のカーソル位置から行右端までのテキスト画面をクリアする。

_iocs_b_era_st 関数は，行左端から現在のカーソル位置までのテキスト画面をクリアする。

_iocs_b_era_al 関数は，カーソルのある行全体をクリアする。

これらの関数は IOCS コール 0x2B を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_iocs_b_format

用　途——ディスクの物理フォーマットを行う。

書　式——#include <sys/iocs.h>

```
int _iocs_b_format (int drive,
                    int recno, int len, const void *addr);
```

解　説——_iocs_b_format 関数はディスクの物理フォーマットを行う。

drive には *pda*×256+*mode* を指定する。*drive*，*pda*，*mode*，*recno* については _iocs_b_seek 関数を参照のこと。

len にはバイト数を指定するが，ハードディスクの場合は**インタリーブコード**を指定する。通常，インタリーブコードは 10M バイトドライブの場合は 6 を，20/40M バイトドライブの場合は 1 を指定する。

addr には **ID データ**の先頭アドレスを指定する。ただし，ハードディスクに対しては不要である。このアドレスには，スーパーバイザ領域も指定できるので注意すること。

_iocs_b_format 関数は IOCS コール 0x4D を発行することによって処理される。

戻り値——成功した場合，ハードディスクならば 0 を含む正の値を，フロッピーディスクならば FDC のステータス情報 (負の値もあり得る) を返す。また，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，物理フォーマットに失敗してもエラー終了とはならず，FDC のステータス情報を返す。

返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　格——*XC*

関連項目——_iocs_b_seek，_iocs_b_verify

_iocs_b_ins

用　　途――カーソル表示行の直後に行を追加する。

書　　式――

```
#include <sys/iocs.h>
void _iocs_b_ins (int cnt);
```

解　　説――_iocs_b_ins関数は現在のカーソル行の直後に*cnt*行追加する。ただし，*cnt*に0を指定すると1とみなされる。また，カーソルのX座標は0となる。

_iocs_b_ins関数はIOCSコール0x2Cを発行することによって処理される。

戻り値――なし。

規　　格――*XC*

_iocs_b_intvcs

用　途——割り込みベクタを設定する。

書　式——#include <sys/iocs.h>

```
void *_iocs_b_intvcs (int vector, void *addr);
```

解　説——_iocs_b_intvcs 関数は割り込みベクタの設定を行う。

vector にはベクタ番号を指定し，*addr* には処理アドレスを指定する。0x00 ～ 0xFF は割り込みベクタ処理のアドレス，0x100 ～ 0x1FF は IOCS コールの処理アドレスである。

割り込みベクタルーチンの最後は "rte"，IOCS コール割り込みルーチンの最後は "rts" を指定する。

_iocs_b_intvcs 関数は IOCS コール 0x80 を発行することによって処理される。

戻り値——設定前の処理アドレスを返す。

規　格——*XC*

_iocs_b_keyinp

用　途——キーコードの読み込みを行う。

書　式——#include <sys/iocs.h>

```
int _iocs_b_keyinp (void);
```

解　説——_iocs_b_keyinp 関数はキーコードの読み込みを行う。また，電卓機能が処理される。

_iocs_b_keyinp 関数は IOCS コール 0x00 を発行することによって処理される。

戻り値——スキャンコード×256+内部コードを返す。スキャンコードは，_iocs_bitsns 関数によって指定するキーコードグループを8倍したものに，そのグループ内のキーのビット位置の番号を加えたものである。たとえば‘A’が押された場合，グループは3，ビット位置は6なので，3×8+6 = 0x1E となる。

規　格——*XC*

関連項目——_iocs_bitsns

_iocs_b_keysns

用　途——キーの先行入力の検査を行う。

書　式——
```
#include <sys/iocs.h>
int _iocs_b_keysns (void);
```

解　説——_iocs_b_keysns 関数はキーの先行入力があるかどうかを調べる。また，電卓機能が処理される。

_iocs_b_keysns 関数は IOCS コール 0x01 を発行することによって処理される。

戻り値——0x10000+スキャンコード×256+内部コードを返す。スキャンコードについては，_iocs_b_keyinp 関数を参照のこと。0 の場合は先行入力がないことを示す。

規　格——*XC*

関連項目——_iocs_b_keyinp

_iocs_b_left

用　途——カーソル位置を指定桁数だけ左へ移動する。

書　式——

```
#include <sys/iocs.h>
void _iocs_b_left (int cnt);
```

解　説——_iocs_b_left 関数はカーソル位置を *cnt* 桁左へ移動する。カーソル位置が左端を越える場合でも，右スクロールは行われない。なお，*cnt* に 0 を指定すると 1 とみなされる。

_iocs_b_left 関数は IOCS コール 0x29 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_locate

用　途——カーソル位置を設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_b_locate (int x, int y);
```

解　説——_iocs_b_locate 関数はカーソル位置を x と y で指定した座標に設定する。ただし，x に−1 を指定した場合は現在のカーソル位置の通知を行う。

_iocs_b_locate 関数は IOCS コール 0x23 を発行することによって処理される。

戻り値——上位 16 ビットに設定前のカーソル位置の X 座標を返し，下位 16 ビットに Y 座標を返す。

規　格——*XC*

_iocs_b_lpeek

用　途――メモリから１ロングワードデータ読み込む。

書　式――#include <sys/iocs.h>

```
int _iocs_b_lpeek (const void *addr);
```

解　説――_iocs_b_lpeek 関数は，*addr*で指定したアドレスから１ロングワードデータ読み込む。主にユーザモードでスーパーバイザエリアをアクセスするために使用する。ただし，**X68000** では *addr*で指定するアドレスは，偶数アドレスでなければならない。

_iocs_b_lpeek 関数は IOCS コール 0x84 を発行することによって処理される。

戻り値――読み込んだロングワードデータを返す。

規　格――*XC*

_iocs_b_lpoke

B

用　途——メモリに1ロングワードデータ書き込む。

書　式——#include <sys/iocs.h>

```
void _iocs_b_lpoke (void *addr, int data);
```

解　説——_iocs_b_lpoke 関数は *addr* で指定したアドレスに *data* で指定した1ロングワードデータを書き込む。主にユーザモードでスーパーバイザエリアをアクセスするために使用する。ただし，**X68000** では *addr* で指定するアドレスは，偶数アドレスでなければならない。

_iocs_b_lpoke 関数は IOCS コール 0x88 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_memset

用　　途——メモリへデータを書き込む。

書　　式——#include <sys/iocs.h>

```
void _iocs_b_memset (void *addr, const void *buff, int len);
```

解　　説——_iocs_b_memset 関数は，*buff* で指定したバッファから *len* で指定したバイト数のデータを *addr* で指定したアドレスへ転送する。ただし，*len* には転送するバイト数−1を指定する。

_iocs_b_memset 関数は IOCS コール 0x89 を発行することによって処理される。

戻 り 値——なし。

注　　意——転送元と転送先の領域が重なってはいけない。

規　　格——*XC*

_iocs_b_memstr

用　途――メモリからデータを読み込む。

書　式――

```
#include <sys/iocs.h>
void _iocs_b_memstr (const void *addr, void *buff, int len);
```

解　説――_iocs_b_memstr 関数は，*addr* で指定したアドレスから *len* で指定したバイト数のデータを読み込み，結果を *buff* で指定したバッファへ格納する。ただし，*len* には読み込むバイト数−1 を指定する。

_iocs_b_memstr 関数は IOCS コール 0x85 を発行することによって処理される。

戻り値――なし。

注　意――転送元と転送先の領域が重なってはいけない。

規　格――*XC*

_iocs_b_print

用　途——文字列を表示する。

書　式——

```
#include <sys/iocs.h>
int _iocs_b_print (const char *message);
```

解　説——_iocs_b_print 関数は *message* で指定した文字列を表示する。*message* で指定する文字列は null 文字で終了していること。

_iocs_b_print 関数は IOCS コール 0x21 を発行することによって処理される。

戻り値——上位 16 ビットに表示後のカーソル位置の X 座標を返し，下位 16 ビットに Y 座標を返す。

規　格——*XC*

_iocs_b_putc

用　途——1 文字表示する。

書　式——

```
#include <sys/iocs.h>
int _iocs_b_putc (int code);
```

解　説——_iocs_b_putc 関数は *code* で指定した 1 文字 (1 バイト) を表示する。

_iocs_b_putc 関数は IOCS コール 0x20 を発行することによって処理される。

戻り値——上位 16 ビットに表示後のカーソル位置の X 座標を返し，下位 16 ビットに Y 座標を返す。

規　格——*XC*

_iocs_b_putmes

用　途――表示位置を指定して文字列を表示する。

書　式――

```
#include <sys/iocs.h>
int _iocs_b_putmes (int color, int x, int y,
                    int max, const char *message);
```

解　説――_iocs_b_putmes 関数は，*x*と *y*で指定した座標から *color*で指定した属性で，*message*で指定した文字列を表示する。

表示される文字列が *max*で指定した文字数よりも長い場合でも，*max*で指定した文字数以上は表示されないが，このとき表示が漢字の前半で打ち切られる場合はその全角文字は表示されず，スペースに置換される。逆に，表示される文字列が *max*で指定した値よりも短い場合，残りはスペースが表示される。

_iocs_b_putmes 関数は本来ファンクション行の表示を行うためのものなので，_iocs_b_consol 関数による表示範囲指定の影響は受けない。また，表示後のカーソル位置も変化しない。

*color*に指定できる値は次のとおり。

- 0　黒
- 1　水色
- 2　黄色
- 3　白
- 4　黒
- 5　水色の強調
- 6　黄色の強調
- 7　白の強調
- 8　黒
- 9　水色のリバース
- 10　黄色のリバース
- 11　白のリバース
- 12　黒
- 13　水色の強調，リバース
- 14　黄色の強調，リバース
- 15　白の強調，リバース

x，*y*にはそれぞれ表示位置を絶対座標で指定し，*max*には表示文字数−1 をバイト単位 (半角文字単位) で指定する。また，*message*には表示する文字列を指定する。ただし，この文字列は null 文字で終了していること。

_iocs_b_putmes 関数は IOCS コール 0x2F を発行することによって処理される。

戻り値——次の X 座標を返す。

規　格——*XC*

関連項目——_iocs_b_console

_iocs_b_read

用　　途――ディスクの読み込みを行う。

書　　式――#include <sys/iocs.h>

```
int _iocs_b_read (int drive,
                  int recno, int len, void *addr);
```

解　　説――_iocs_b_read 関数は，*drive* で指定したドライブの *recno* で指定したレコードから，*len* で指定したバイト数を読み込み，結果を *addr* で指定した領域に格納する。

drive と *recno* については _iocs_b_seek 関数を参照のこと。また，*len* については _iocs_b_verify 関数を参照のこと。

addr には読み込みバッファの先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_read 関数は IOCS コール 0x46 を発行することによって処理される。

戻 り 値――成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，読み込みに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　　格――*XC*

関連項目――_iocs_b_seek，_iocs_b_verify

_iocs_b_readdi

用　途——フロッピーディスクの診断のための読み込みを行う。

書　式——

```
#include <sys/iocs.h>
int _iocs_b_readdi (int drive,
                    int recno, int len, void *addr);
```

解　説——_iocs_b_readdi 関数は，*drive*で指定したドライブの *recno*で指定したレコードからのデータを，*len*で指定したバイト数を診断のために読み込み，結果を *addr*で指定した領域に格納する。

*drive*と *recno*については_iocs_b_seek 関数を参照のこと。

*len*には 1024 の倍数を指定し，*addr*には読み込みバッファの先頭アドレスを指定する。ただし，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_readdi 関数は IOCS コール 0x42 を発行することによって処理される。

戻り値——成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

読み込みに失敗してもエラー終了とはならず，必ず指定バイト数の読み込みを行う。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　格——*XC*

関連項目——_iocs_b_seek, _iocs_b_verify

_iocs_b_readdl

用　　途――フロッピーディスクの削除データを読み込む。

書　　式――#include <sys/iocs.h>

```
int _iocs_b_readdl (int drive,
                    int recno, int len, void *addr);
```

解　　説――_iocs_b_readdl 関数は，*drive*で指定したドライブの *recno*で指定したレコードから，*len*で指定したバイト数の削除データを読み込み，結果を *addr*で指定した領域に格納する。

*drive*と *recno*については_iocs_b_seek 関数を参照のこと。また，*len*については_iocs_b_verify 関数を参照のこと。

*addr*には読み込みバッファの先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_readdl 関数は IOCS コール 0x4C を発行することによって処理される。

戻 り 値――成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，読み込みに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　　格――*XC*

関連項目――_iocs_b_seek，_iocs_b_verify

_iocs_b_readid

用　途——フロッピーディスクの ID データを読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_b_readid (int drive, int recno, void *addr);
```

解　説——_iocs_b_readid 関数は，*drive* で指定したドライブの *recno* で指定したレコードから，ID データを読み込み，結果を *addr* で指定した領域に格納する。*drive* と *recno* については _iocs_b_seek 関数を参照のこと。

addr には読み込みバッファの先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_readid 関数は IOCS コール 0x4A を発行することによって処理される。

戻り値——成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF?? を返す。

フロッピーディスクの場合，読み込みに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　格——*XC*

関連項目——_iocs_b_readdl

_iocs_b_recali

用　途——トラック0へシークする。

書　式——#include <sys/iocs.h>

```
int _iocs_b_recali(int drive);
```

解　説——_iocs_b_recali 関数は，*drive* で指定したドライブのヘッドを0トラックへシークさせる。

drive には *pda*×256 を指定する。ただし，フロッピーディスクの場合，*drive* に *pda*×256+255 を指定すると，強制レディ状態での調査を行う。*pda* については _iocs_b_seek 関数を参照のこと。

_iocs_b_recali 関数は IOCS コール 0x47 を発行することによって処理される。

戻り値——成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，シークに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_seek 関数を参照のこと。

また，フロッピーディスクに対して強制レディ状態で調査を行った場合は，指定ドライブが存在するかどうかをチェックすることができる。この場合，戻り値のリザルトステータス0のビット4 (EQUIPMENT CHECK) が1であれば，指定したドライブが存在しないことを示す。

規　格——*XC*

関連項目——_iocs_b_seek

_iocs_b_right

用　途——カーソル位置を指定桁数だけ右へ移動する。

書　式——#include <sys/iocs.h>

```
void _iocs_b_right (int cnt);
```

解　説——_iocs_b_right 関数はカーソル位置を *cnt* 桁右へ移動する。カーソル位置が右端を越える場合でも，左スクロールは行われない。なお，*cnt* に 0 を指定すると 1 とみなされる。

_iocs_b_right 関数は IOCS コール 0x28 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_seek

用　途——指定トラックまでシークする。

書　式——#include <sys/iocs.h>

```
int _iocs_b_seek (int drive, int recno);
```

解　説——_iocs_b_seek 関数は *drive* で指定したドライブのヘッドを，*recno* で指定したレコードのトラックまでシークさせる。

drive には *pda*×256 +*mode* を指定する。*pda* の内容は次のとおり。

- 0x80 ～ 0x8F　ハードディスク
- 0x90 ～ 0x93　フロッピーディスク

ハードディスクの場合は，*recno* に 256 バイト単位のレコード番号を指定する。フロッピーディスクの場合は，*recno* にセクタ長，トラック，サイド，セクタの順に上位から 8 ビットずつ指定する。

セクタ長の指定は次のとおり。

- 0　128 バイト
- 1　256 バイト
- 2　512 バイト
- 3　1024 バイト

mode の指定はハードディスクの場合は 0 を指定するが，フロッピーディスクの場合は次のようなビットフィールドを設定する。

0	fm/mfm	retry	seek	0	0	0	0

たとえば，fm/mfm ビットは 0 で FM(単密度)，1 で MFM(倍密度) を意味し，retry ビットは 0 でリトライしないことを，1 でリトライすることを意味する。また，*seek* ビットは 0 でシークしないで実行することを，1 でシークして実行することを意味する。

retry で 1 が指定された場合は 10 回リトライを行い，最初の 5 回はシークなしのリトライ，残り 5 回はシークあり (リキャリブレートシーク) のリトライを行う。

_iocs_b_seek 関数は IOCS コール 0x40 を発行することによって処理される。

戻り値——成功した場合，ハードディスクならば0を含む正の値を，フロッピーディスクならば FDC のステータス情報 (負の値もあり得る) を返す。また，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，シークに失敗してもエラー終了とはならず，FDC のステータス情報として失敗したことを返す。

FDC のステータス情報は次のとおり。

- ビット 31 ～ 24　リザルトステータス 0
- ビット 23 ～ 16　コマンド終了時のシリンダ番号
- ビット 15 ～ 0　不定

リザルトステータス 0 の内容は次のとおり。

ビット 7, 6	割り込み発生要因 00... コマンド正常終了 01... コマンド異常終了 10... 無効なコマンド 11... ドライブ状態の変化
ビット 5	シーク動作時にセット
ビット 4	Fault 信号を受けた場合などにセット
ビット 3	Ready 状態でないときにセット
ビット 2	割り込み発生時のヘッドの状態
ビット 1, 0	割り込み発生時のドライブ番号 00... ドライブ 0 01... ドライブ 1 10... ドライブ 2 11... ドライブ 3

規　格——*XC*

_iocs_b_sftsns

用　途――シフトキーの押下げ状態を検査する。

書　式――#include <sys/iocs.h>

```
int _iocs_b_sftsns (void);
```

解　説――_iocs_b_sftsns 関数はシフトキーの押下げ状態を調べる。

_iocs_b_sftsns 関数は IOCS コール 0x02 を発行することによって処理される。

戻り値――戻り値の下位 15 ビットには各シフトキーの押下げ状態が返されるが，LED のついているキーについては LED の点灯状態を返す。戻り値の下位 15 ビットの内容は次のとおり。いずれも該当ビットが 1 の場合に押下されていることを示す。

15	14	13	12	11	10
未定義	全角	ひらがな	INS	CAPS	コード入力

9	8	7	6	5	4
ローマ字	かな	CAPS	コード入力	ローマ字	かな

3	2	1	0
OPT.2	OPT.1	CTRL	SHIFT

規　格――*XC*

_iocs_b_super

用　途——スーパーバイザモードとユーザモードとを切り替える。

書　式——
```
#include <sys/iocs.h>
int _iocs_b_super (int ssp);
```

解　説——_iocs_b_super 関数は**スーパーバイザモード**と**ユーザモード**との切り替えを行う。*ssp*が 0 の場合はユーザモードからスーパーバイザモードへ，0 以外の場合はスーパーバイザモードからユーザモードへ切り替える。

_iocs_b_super 関数は IOCS コール 0x81 を発行することによって処理される。

戻り値——切り替える前の ssp を返す。ただし，スーパーバイザモードのときに再度スーパーバイザモードに切り替えた場合はエラーとなり，負の値を返す。

規　格——*XC*

_iocs_b_up

用　途——カーソル位置を指定行数だけ上へ移動する。

書　式——
```
#include <sys/iocs.h>
void _iocs_b_up (int cnt);
```

解　説——_iocs_b_up 関数はカーソル位置を *cnt* 行上へ移動する。カーソル位置が先頭行を越える場合でも，スクロールダウンは行われない。なお，*cnt* に 0 を指定すると 1 とみなされる。

_iocs_b_up 関数は IOCS コール 0x26 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_b_up_s

用　　途――カーソル位置を1行上へ移動する。

書　　式――#include <sys/iocs.h>

```
void _iocs_b_up_s (void);
```

解　　説――_iocs_b_up_s 関数はカーソル位置を1行上へ移動する。ただし，現在のカーソル位置が先頭行だった場合は**スクロールダウン**が行われる。

_iocs_b_up_s 関数は IOCS コール 0x25 を発行することによって処理される。

戻 り 値――なし。

規　　格――*XC*

_iocs_b_verify

用　途――データの比較を行う。

書　式――#include <sys/iocs.h>

```
int _iocs_b_verify (int drive, int recno,
                    int len, const void *addr);
```

解　説――_iocs_b_verify 関数は，*drive* で指定したドライブの *recno* で指定したレコードからのデータを，*len* で指定したバイト数読み込み，*addr* で指定した領域の内容と比較する。

drive と *recno* については_iocs_b_seek 関数を参照のこと。

len にはハードディスクの場合 256 の倍数を，フロッピーディスクの場合は 1024 の倍数を指定する。

addr には比較対象のデータの先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_verify 関数は IOCS コール 0x41 を発行することによって処理される。

戻り値――成功した場合，ハードディスクならば 0 を含む正の値を，フロッピーディスクならば FDC のステータス情報 (負の値もあり得る) を返す。また，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，比較に失敗してもエラー終了とはならず，FDC のステータス情報として失敗したことを返す。

FDC のステータス情報は次のとおり。

- ビット 31 ～ 24　リザルトステータス 0
- ビット 23 ～ 16　リザルトステータス 1
- ビット 15 ～ 8　リザルトステータス 2
- ビット 7 ～ 0　シリンダ番号

リザルトステータス0の内容は次のとおり。

ビット7, 6	割り込み発生要因 00... コマンド正常終了 01... コマンド異常終了 10... 無効なコマンド 11... ドライブ状態の変化
ビット5	シーク動作時にセット
ビット4	Fault 信号を受けた場合などにセット
ビット3	Ready 状態でないときにセット
ビット2	割り込み発生時のヘッドの状態
ビット1, 0	割り込み発生時のドライブ番号 00... ドライブ0 01... ドライブ1 10... ドライブ2 11... ドライブ3

リザルトステータス1の内容は次のとおり。

ビット7	最終セクタを越えてアクセスした場合にセット
ビット6	つねに0
ビット5	ID または CRC エラーのときにセット
ビット4	MPU または DMA が，FDC に対して規定時間内にデータの入出力ができなかった場合にセット
ビット3	つねに0
ビット2	セクタが見つからない場合にセット
ビット1	書き込み時にライトプロテクトがかかっていた場合にセット
ビット0	セクタが存在しない，あるいはマークのみでデータフィールドがない場合にセット

リザルトステータス2の内容は次のとおり。

ビット7	つねに0
ビット6	削除データ読み込み時に通常データを読んだ場合，またはその逆の場合にセット
ビット5	CRCエラーのときにセット
ビット4	指定したトラックが見つからない場合にセット。このビットがセットされ，リザルトステータス1のビット2もセットされる場合は，トラックが見つからなかったことを示す
ビット3	ベリファイでデータが一致した場合にセット
ビット2	ベリファイでデータが一致しなかった場合にセット
ビット1	指定したトラック番号が規定外の場合にセット
ビット0	マークのみでデータフィールドがなかったために，リザルトステータス1のビット0がセットされた場合にセット

規　　格——*XC*

関連項目——_iocs_b_seek

_iocs_b_wpeek

用　途——メモリから１ワードデータ読み込む。

書　式——

```
#include <sys/iocs.h>
int _iocs_b_wpeek (unsigned short *addr);
```

解　説——_iocs_b_wpeek 関数は，*addr*で指定したアドレスから１ワードデータ読み込む。主にユーザモードでスーパーバイザエリアをアクセスするために使用する。ただし，**X68000** では *addr*で指定するアドレスは偶数アドレスでなければならない。

_iocs_b_wpeek 関数は IOCS コール 0x83 を発行することによって処理される。

戻り値——読み込んだワードデータを返す。

規　格——*XC*

_iocs_b_wpoke

用　　途——メモリに1ワードデータ書き込む。

書　　式——#include <sys/iocs.h>

```
void _iocs_b_wpoke (void *addr, int data);
```

解　　説——_iocs_b_wpoke 関数は，*addr*で指定したアドレスへ*data*で指定した1ワードデータを書き込む。主にユーザモードでスーパーバイザエリアをアクセスするために使用する。ただし，**X68000** では *addr*で指定するアドレスは偶数アドレスでなければならない。

_iocs_b_wpoke 関数はIOCSコール 0x87 を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_iocs_b_write

用　途——ディスクにデータを書き込む。

書　式——

```
#include <sys/iocs.h>
int _iocs_b_write (int drive, int recno,
                   int len, const void *addr);
```

解　説——_iocs_b_write 関数は，*addr* で指定した領域のデータを *len* で指定したバイト数だけ，*drive* で指定したドライブの *recno* で指定したレコードから書き込む。

drive と *recno* については_iocs_b_seek 関数を参照のこと。また，*len* については_iocs_b_verify 関数を参照のこと。

addr には書き込みデータの先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_b_write 関数は IOCS コール 0x45 を発行することによって処理される。

戻り値——成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，書き込みに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

注　意——_iocs_b_write 関数は，**Human68k** のバッファリング処理とは無関係にディスクに対して書き込みを行う。したがって不用意な書き込みは，ディスクと **Human68k** のバッファデータとの整合性を損ねるので注意すること。

規　格——*XC*

関連項目——_iocs_b_seek，_iocs_b_verify

_iocs_b_writed

用　　途——フロッピーディスクへ削除データを書き込む。

書　　式——#include <sys/iocs.h>

```
int _iocs_b_writed (int drive, int recno,
                    int len, const void *addr);
```

解　　説——_iocs_b_writed 関数はフロッピーディスクに対して削除データを書き込む。

drive, *recno*, *len*, *addr* の各引数については，_iocs_b_write 関数を参照のこと。

_iocs_b_writed 関数は IOCS コール 0x49 を発行することによって処理される。

戻 り 値——成功した場合は，FDC のステータス情報 (負の値もあり得る) を返し，失敗した場合は 0xFFFFFF??を返す。

フロッピーディスクの場合，書き込みに失敗してもエラー終了とはならず，FDC のステータス情報としてエラーが発生したことを返す。返される FDC のステータス情報については，_iocs_b_verify 関数を参照のこと。

規　　格——*XC*

関連項目——_iocs_b_verify, _iocs_b_write

_iocs_bgctrlgt

用　途——バックグラウンドコントロールレジスタを読み込む。

書　式——

```
#include <sys/iocs.h>
int _iocs_bgctrlgt (int mode);
```

解　説——_iocs_bgctrlgt 関数は，バックグラウンド (BG) コントロールレジスタの読み込みを行う。*mode* には BG コントロールレジスタ (0 ～ 1) の番号を指定する。

_iocs_bgctrlgt 関数は IOCS コール 0xCB を発行することによって処理される。

戻り値——成功した場合はテキストページ×2+表示モード (0 ～ 1) を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格——*XC*

_iocs_bgctrlst

用　　途——バックグラウンドコントロールレジスタを設定する。

書　　式——
```
#include <sys/iocs.h>
int _iocs_bgctrlst (int mode, int page, int onoff);
```

解　　説——`_iocs_bgctrlst`関数は，バックグラウンド(BG)コントロールレジスタの設定を行う。*mode*にはBGコントロールレジスタ(0～1)の番号を指定する。

*page*にはテキストページ(0～1)を指定し，*onoff*には表示のオン/オフを指定する。表示をオンにするには1を，オフにするには0を指定すること。ただし，−1が指定された場合は設定を変更せずに，現在の設定を使用する。

`_iocs_bgctrlst`関数はIOCSコール`0xCA`を発行することによって処理される。

戻り値——成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー(画面サイズが768 × 512のときなど)

規　　格——*XC*

_iocs_bgscrlgt

用　　途——バックグラウンドスクロールレジスタを読み込む。

書　　式——#include <sys/iocs.h>

```
int _iocs_bgscrlgt (int mode, int *x, int *y);
```

解　　説——_iocs_bgscrlgt 関数はバックグラウンドスクロールレジスタの読み込みを行う。

mode にはスクロールレジスタの番号を指定する。

x，*y* にはそれぞれ X 座標，Y 座標を格納する領域のポインタを指定する。返される値は 0 ～ 1023 の範囲である。

_iocs_bgscrlgt 関数は IOCS コール 0xC9 を発行することによって処理される。

戻 り 値——成功した場合は 0 を返し，*x*，*y* で指定したポインタに X 座標，Y 座標が格納される。失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　　格——*XC*

_iocs_bgscrlst

用　途――バックグラウンドスクロールレジスタを設定する。

書　式――#include <sys/iocs.h>

```
int _iocs_bgscrlst (int mode, int x, int y);
```

解　説――_iocs_bgscrlst 関数はバックグラウンド(BG)スクロールレジスタの設定を行う。

*mode*には次の値を指定する。

ビット31	0...垂直帰線期間を検出してから設定する 1...垂直帰線期間を検出しないで設定する
ビット0	0...BG 0 を設定する 1...BG 1 を設定する

x, *y*にはそれぞれX座標，Y座標を0～1023の範囲で指定するが，いずれも−1が指定された場合は，設定を変更せずに現在の設定を使用する。

_iocs_bgscrlst 関数はIOCS コール 0xC8 を発行することによって処理される。

戻り値――成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー(画面サイズが768 × 512のときなど)

規　格――*XC*

_iocs_bgtextcl

用　途——バックグラウンドテキストをクリアする。

書　式——#include <sys/iocs.h>

```
int _iocs_bgtextcl (int page, int code);
```

解　説——_iocs_bgtextcl 関数はバックグラウンド (BG) テキストのクリアを行う。

page にはクリアするテキストページ (0 ～ 1) を指定し，*code* にはパターンコードを指定する。このパターンコードの値は次のとおり。

ビット 15	0 . . . 縦方向反転しない 1 . . . 縦方向反転する
ビット 14	0 . . . 横方向反転しない 1 . . . 横方向反転する
ビット 8 ～ 11	パレットブロック (0 ～ 15)
ビット 0 ～ 7	PCG コード (0 ～ 255)
上記以外	つねに 0

たとえば *code* に 0x873C を指定した場合，PCG コードは 60，パレットブロックは 7，縦方向のみ反転となる。

_iocs_bgtextcl 関数は IOCS コール 0xCC を発行することによって処理される。

戻り値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格——*XC*

_iocs_bgtextgt

用　途——バックグラウンドテキストを読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_bgtextgt (int page, int x, int y);
```

解　説——_iocs_bgtextgt 関数はバックグラウンドテキストの読み込みを行う。

page にはテキストページ (0 ~ 1) を指定し，*x*，*y* にはそれぞれテキスト X 座標，テキスト Y 座標を 0 ~ 63 の範囲で指定する。

_iocs_bgtextgt 関数は IOCS コール 0xCE を発行することによって処理される。

戻り値——成功した場合はパターンコードを返し，失敗した場合は次の値を返す。パターンコードについては，_iocs_bgtextcl 関数を参照のこと。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格——*XC*

関連項目——_iocs_bgtextcl

_iocs_bgtextst

用　途――バックグラウンドテキストを設定する。

書　式――#include <sys/iocs.h>

```
int _iocs_bgtextst (int page, int x, int y, int code);
```

解　説――_iocs_bgtextst 関数はバックグラウンドテキストの設定を行う。

*page*にはテキストページ(0～1)を指定し，*x*，*y*にはそれぞれテキストX座標，テキストY座標を0～63の範囲で指定する。

*code*にはパターンコードを指定する。パターンコードの内容は_iocs_bgtextcl関数を参照のこと。

_iocs_bgtextst 関数はIOCSコール 0xCD を発行することによって処理される。

戻り値――成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー(画面サイズが768×512のときなど)

規　格――*XC*

関連項目――_iocs_bgtextcl

_iocs_bindatebcd

用　途――2進数の日付を内部時計にセットできる形式に変換する。

書　式――#include <sys/iocs.h>

```
int _iocs_bindatebcd (int date);
```

解　説――_iocs_bindatebcd 関数は，*date*で指定した2進数形式の日付データを，_iocs_bindateset 関数で使用できる BCD 形式に変換する。

*date*に指定するビットフィールドの値は次のとおり。

```
0000yyyy yyyyyyyy mmmmmmmm dddddddd
```

- dddddddd　　日 (01 ～ 31)
- mmmmmmmm　　月 (01 ～ 12)
- yyyyyyyyyyyy　　年 (1980 ～ 2079)

_iocs_bindatebcd 関数は IOCS コール 0x50 を発行することによって処理される。

戻り値――成功した場合は次のようなビットフィールドを返し，失敗した場合は−1を返す。年/月/日はいずれも BCD2 桁である。

```
UUUUWWWW YYYYYYYY MMMMMMMM DDDDDDDD
```

- DDDDDDDD　　日 (01 ～ 31)
- MMMMMMMM　　月 (01 ～ 12)
- YYYYYYYY　　年 (00 ～ 99，1980 年からの相対年数)
- WWWW　　曜日カウンタ (0 ～ 6，0 で日曜日)
- UUUU　　閏年カウンタ (0 ～ 3)

規　格――*XC*

関連項目――_iocs_bindateset

_iocs_bindateget

用　途——内部時計から日付を読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_bindateget (void);
```

解　説——_iocs_bindateget 関数は，内部時計から現在の日付を読み込む。

_iocs_bindateget 関数は IOCS コール 0x54 を発行することによって処理される。

戻り値——次のようなビットフィールドを返す。年/月/日はいずれも BCD2 桁である。

```
0000WWWW YYYYYYYY MMMMMMMM DDDDDDDD
```

- DDDDDDDD　日 (01 ～ 31)
- MMMMMMMM　月 (01 ～ 12)
- YYYYYYYY　年 (00 ～ 99，1980 年からの相対年数)
- WWWW　曜日 (0 ～ 6，0 で日曜日)

規　格——*XC*

_iocs_bindateset

用　　途——内部時計に日付を設定する。

書　　式——#include <sys/iocs.h>

```
void _iocs_bindateset (int date);
```

解　　説——_iocs_bindateset関数は*date*で指定したBCDの日付を内部時計に設定する。*date*の内容については_iocs_bindatebcd関数の戻り値を参照のこと。

_iocs_bindateset関数はIOCSコール0x51を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

関連項目——_iocs_bindatebcd

_iocs_bitsns

用　途——指定キーの押下げ状態を検査する。

書　式——

```
#include <sys/iocs.h>
int _iocs_bitsns (int group);
```

解　説——_iocs_bitsns 関数は *group* で指定したキーコードグループの押下げ状態を調べる。

キーコードグループは次のとおり。縦軸がキーコードグループ，横軸がビット位置を表す。

	0	1	2	3	4	5	6	7
0	未定義	ESC	1 !	2 "	3 #	4 $	5 %	6 &
1	7 '	8 (	9)	0	- =	^	\ |	BS
2	TAB	Q	W	E	R	T	Y	U
3	I	O	P	@	[	CR	A	S
4	D	F	G	H	J	K	L	;
5	:	]	Z	X	C	V	B	N
6	M	, <	. >	/ ?	_	SPC	HOME	DEL
7	ROLLUP	ROLLDN	UNDO	←	↑	→	↓	CLR
8	/	*	-	7	8	9	+	4
9	5	6	=	1	2	3	ENTER	0
A	,	.	記号	登録	HELP	XF1	XF2	XF3
B	XF4	XF5	かな	ローマ	コード	CAPS	INS	ひら
C	全角	BREAK	COPY	F1	F2	F3	F4	F5
D	F6	F7	F8	F9	F10	未定義	未定義	未定義
E	SHIFT	CTRL	OPT.1	OPT.2	未定義	未定義	未定義	未定義
F	未定義	未定義	未定義	未定義	未定義	未定義	未定義	未定義

_iocs_bitsns 関数は IOCS コール 0x04 を発行することによって処理される。

戻り値——指定したキーコードグループの押下げ状態を返す。該当ビットが 1 ならば，そのキーが押されていることを示している。

規　格——*XC*

_iocs_bootinf

用　途——パワーON情報とシステムのブート情報を返す。

書　式——#include <sys/iocs.h>

```
int _iocs_bootinf (void);
```

解　説——_iocs_bootinf 関数はパワー ON 情報とシステムのブート情報を返す。

_iocs_bootinf 関数は IOCS コール 0x8E を発行することによって処理される。

戻り値——ブート情報を返す。ブート情報の内容は次のとおり。

ビット 31 ～ 24	パワー ON 情報 0x00 ... パワースイッチにより起動 0x01 ... 外部スイッチにより起動 0x02 ... タイマにより起動
ビット 23 ～ 0	システムのブート情報 0x000080 ～ 0x00008f ハードディスクから起動 (下位 4 ビットがブートしたドライブ番号) 0x000090 ～ 0x000093 フロッピーディスクから起動 (下位 2 ビットがブートしたドライブ番号) 0xED0000 ～ 0xED3FFF S–RAM からブートしたときのブートアドレス 上記以外の値 ROM からブートしたときのブートアドレス

規　格——*XC*

B

_iocs_box

用　　途——グラフィック画面にボックスを描画する。

書　　式——#include <sys/iocs.h>

```
int _iocs_box (const struct _boxptr *ptr);
```

解　　説——_iocs_box 関数はグラフィック画面にボックスを描画する。*ptr* に指定する boxptr 構造体は次のとおり。

```
struct _boxptr {
    short x1;                  /* 始点座標の X */
    short y1;                  /* 始点座標の Y */
    short x2;                  /* 終点座標の X */
    short y2;                  /* 終点座標の Y */
    unsigned short color;      /* パレットコード */
    unsigned short linestyle;  /* ラインスタイル */
};
```

_iocs_box 関数は IOCS コール 0xB9 を発行することによって処理される。

戻 り 値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　　格——*XC*

_iocs_circle

用　途――グラフィック画面に円を描画する。

書　式――#include <sys/iocs.h>

```
int _iocs_circle (const struct _circleptr *ptr);
```

解　説――_iocs_circle 関数はグラフィック画面に円を描画する。*ptr* で指定する _circleptr 構造体は次のとおり。

```
struct _circleptr {
    short x;                /* 中心の X 座標 */
    short y;                /* 中心の Y 座標 */
    unsigned short radius;  /* 半径 */
    unsigned short color;   /* パレットコード */
    short start;            /* 円弧開始角度 */
    short end;              /* 円弧終了角度 */
    unsigned short ratio;   /* 比率 */
};
```

円弧開始角度/円弧終了角度は 0 ～ 360 の値を指定する。ただし，範囲外の角度を指定した場合は 360 を指定したとみなされる。また，角度に負の値を指定すると円弧のかわりに扇型を描画する。このとき角度の指定は絶対値を取られる。

比率は 0 ～ 65535 までの値を指定する。比率が 0 ～ 255 のとき，半径は横方向の半径を意味し，縦方向との比率は X：256 となる。また，比率が 256 ～ 65535 のとき，半径は縦方向の半径を意味し，横方向との比率は X：256 となる。

_iocs_circle 関数は IOCS コール 0xBB を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格――*XC*

C

_iocs_clipput

用　途――テキスト画面にパターンを書き出す (クリッピング処理つき)。

書　式――#include <sys/iocs.h>

```
void _iocs_clipput (int x, int y, const struct _fntbuf *buff,
                          const struct _clipxy *clipptr);
```

解　説――_iocs_clipput 関数は *x* および *y* で指定したテキスト画面の座標へ，*buff* で指定したパターンデータを書き込むが，このとき *clipptr* の内容によってクリッピング処理を行う。

buff にはスーパーバイザ空間も指定できるので，誤ったアドレスを指定してはならない。また，*buff* に指定する _fntbuf 構造体には書き込む X 方向，Y 方向のドット数を short で指定する。*buff* に指定する _fntbuf 構造体は次のとおり。

```
struct _fntbuf {
    short xl;                  /* 処理する X 方向のドット数 */
    short yl;                  /* 処理する Y 方向のドット数 */
    unsigned char buffer[72];  /* 読み込みバッファ */
};
```

clipptr に指定する _clipxy 構造体には，表示領域のクリッピング座標を指定する。なお，最終位置を指定するメンバ xe および ye には，指定する座標に 1 を足した値を指定する。*clipptr* に指定する _clipxy 構造体は次のとおり。

```
struct _clipxy {
    short xs; /* 表示領域の先頭の X 座標 */
    short ys; /* 表示領域の先頭の Y 座標 */
    short xe; /* 表示領域の最終の X 座標 */
    short ye; /* 表示領域の最終の Y 座標 */
};
```

_iocs_clipput 関数は IOCS コール 0x1C を発行することによって処理される。

戻り値――なし。

注　意――もし，_fntbuf 構造体に入らないサイズを指定する場合は，独自に malloc 関数などによって領域を確保してから使用すること。

規　格――*XC*

関連項目――_iocs_textget，_iocs_textput，malloc

_iocs_contrast

用　途――画面のコントラストを設定する。

書　式――#include <sys/iocs.h>

```
int _iocs_contrast (int mode);
```

解　説――_iocs_contrast 関数は画面のコントラストを *mode* で指定した値に設定する。

mode に 0 ~ 15 を指定した場合は，画面をその値のコントラストに設定するが，−1 を指定した場合は現在のコントラストを戻り値に設定し，−2 を指定した場合はコントラストをシステム規定値に設定する。

_iocs_contrast 関数は IOCS コール 0x11 を発行することによって処理される。

戻り値――設定変更前の画面のコントラストを返す。

規　格――*XC*

_iocs_crtcras

用　途――CRTC のラスタ割り込みを設定する。

書　式――#include <sys/iocs.h>

```
int _iocs_crtcras (const void *addr, int raster);
```

解　説――_iocs_crtcras 関数は CRTC に設定したラスタ割り込みの処理を設定する。*addr* には割り込み処理アドレスを指定するが，0 を指定することで割り込みを禁止できる。また，*raster* には割り込みを発生させるラスタを指定する。

_iocs_crtcras 関数は IOCS コール 0x6D を発行することによって処理される。

戻り値――割り込みが設定された場合は 0 を返し，すでに使用している場合は 0 以外の値を返す。

規　格――*XC*

_iocs_crtmod

用　　途――画面モードを設定する。

書　　式――
```
#include <sys/iocs.h>
int _iocs_crtmod (int mode);
```

解　　説――_iocs_crtmod 関数は *mode* で指定した画面モードを設定する。

このとき，テキスト画面は初期状態 (画面はクリアされ，パレットはデフォルト) となり，グラフィック画面，スプライト，バックグラウンド (BG) 画面は無表示状態 (初期化はされない) となる。

mode に−1 を指定すると，現在の画面モードを返す。

mode に 0x100 以上の値を指定すると，下位 8 ビットの画面モードに切り替えるだけで，画面の初期化 (画面のクリア，パレット，コントラスト，表示モードの初期化) は行わない。

mode に指定できる値は次のとおり。

mode	周波数	解像度	発色数	VRAM
0	31kHz	512×512	16/16	1024
1	15kHz	512×512	16/16	1024
2	31kHz	256×256	16/16	1024
3	15kHz	256×256	16/16	1024
4	31kHz	512×512	16/16	512
5	15kHz	512×512	16/16	512
6	31kHz	256×256	16/16	512
7	15kHz	256×256	16/16	512
8	31kHz	512×512	16/256	512
9	15kHz	512×512	16/256	512
10	31kHz	256×256	16/256	512
11	15kHz	256×256	16/256	512
12	31kHz	512×512	16/65536	512
13	15kHz	512×512	16/65536	512
14	31kHz	256×256	16/65536	512
15	15kHz	256×256	16/65536	512
16	31kHz	768×512	16/16	1024
17	24kHz	1024×424	16/16	1024
18	24kHz	1024×848	16/16	1024

_iocs_crtmod 関数は IOCS コール 0x10 を発行することによって処理される。

戻り値——*mode*に-1を指定したときだけ，現在の画面モードを返す。

規　　格——*XC*

_iocs_dakjob

用　　途――濁点処理を行う。

書　　式――#include <sys/iocs.h>

```
int _iocs_dakjob (char *end);
```

解　　説――_iocs_dakjob 関数は**濁点処理**を行う。

*end*には文字列の終端のnull文字のアドレスを指定する。最後の全角文字に濁点処理ができないときは文字列に‘ﾞ’を追加するため，null文字の後に3バイト分の領域が必要である。また，‘ﾞ’を追加した場合は‘ﾞ’の後にnull文字を付加するので，文字列の途中を指定すると‘ﾞ’の後の文字列が切れてしまう。したがって，文字列の途中は指定しないこと。

_iocs_dakjob 関数はIOCSコール 0xA4 を発行することによって処理される。

戻り値――変換ステータスを返す。変換ステータスは次のとおり。

- 0　最後の全角文字に濁点処理をした
- 2　‘ﾞ’を付加した

規　　格――*XC*

関連項目――_iocs_hanjob

_iocs_dateasc

用　途——日付を表す 2 進数形式のデータを文字列に変換する。

書　式——

```
#include <sys/iocs.h>
int _iocs_dateasc (int date, char *addr);
```

解　説——_iocs_dateasc 関数は，*date* で指定した 2 進数形式の日付データを文字列に変換し，*addr* で指定した領域へ格納する。

date に指定するビットフィールドの内容は次のとおり。

```
FFFFyyyy yyyyyyyy mmmmmmmm dddddddd
```

- dddddddd　日 (01 〜 31)
- mmmmmmmm　月 (01 〜 12)
- yyyyyyyyyyyy　年 (1980 〜 2079)

FFFF には文字列の変換形式を設定する。

- 0　1993/06/23
- 1　1993–06–23
- 2　93/06/23
- 3　93–06–23

addr には文字列を格納するバッファの先頭アドレスを指定するが，このバッファは最低 11 バイト以上の領域を確保しなければならない。また，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_dateasc 関数は IOCS コール 0x5A を発行することによって処理される。

戻り値——失敗した場合は–1 を返す。

規　格——*XC*

_iocs_datebin

用　途——日付の形式をBCDから2進数に変換する。

書　式——#include <sys/iocs.h>

```
int _iocs_datebin (int date);
```

解　説——_iocs_datebin 関数は *date* で指定したBCD形式の日付を2進数形式に変換する。

date に指定するビットフィールドの内容は次のとおり。年/月/日はいずれもBCD2桁である。

```
0000 WWWW YYYYYYYY MMMMMMMM DDDDDDDD
```

- DDDDDDDD　日 (01 ～ 31)
- MMMMMMMM　月 (01 ～ 12)
- YYYYYYYY　年 (00 ～ 99, 1980年からの相対年数)
- WWWW　曜日 (0 ～ 6, 0で日曜日)

_iocs_datebin 関数はIOCSコール 0x55 を発行することによって処理される。

戻り値——次のようなビットフィールドを返す。

```
wwwwyyyy yyyyyyyy mmmmmmmm dddddddd
```

- dddddddd　日 (01 ～ 31)
- mmmmmmmm　月 (01 ～ 12)
- yyyyyyyyyyyy　年 (1980 ～ 2079)
- wwww　曜日 (0 ～ 6, 0で日曜日)

規　格——*XC*

_iocs_datecnv

用　途——日付を表す文字列を 2 進数形式に変換する。

書　式——

```
#include <sys/iocs.h>
int _iocs_datecnv (const char *addr);
```

解　説——_iocs_datecnv 関数は，*addr* で指定した文字列を 2 進数形式の日付データへ変換する。

addr には日付を表す文字列の先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

また，文字列は “1993/06/23” の形式で指定する。日付の区切り記号は “/” でも “-” でも，それ以外でもよい。

_iocs_datecnv 関数は IOCS コール 0x58 を発行することによって処理される。

戻り値——成功した場合は，次のようなビットフィールドを返し，失敗した場合は−1 を返す。ただし，閏年や大小の月の判定は行わない。

```
0000yyyy yyyyyyyy mmmmmmmm dddddddd
```

- dddddddd　　日 (01 ～ 31)
- mmmmmmmm　月 (01 ～ 12)
- yyyyyyyyyyyy　年 (1980 ～ 2079)

規　格——*XC*

_iocs_dayasc

用　　途――曜日を表す2進数のデータを文字列に変換する。

書　　式――#include <sys/iocs.h>

```
void _iocs_dayasc (int day, char *addr);
```

解　　説――_iocs_dayasc 関数は，*day*で指定した2進数形式の曜日データを文字列に変換し，*addr*で指定した領域へ格納する。

*day*には0～6を指定する。この範囲外の値を指定すると，8で割ったあまりが使用される(つねに0～7を使用する)。

*addr*には文字列を格納するバッファの先頭アドレスを指定するが，このバッファは最低3バイト以上の領域を確保しなければならない。また，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_dayasc 関数はIOCSコール 0x5C を発行することによって処理される。

戻り値――なし。

*addr*で示されるバッファには“月”，“火”，“水”，“木”，“金”，“土”，“日”，“?”のいずれかが格納される。“?”が格納された場合は*day*に7を指定したか，指定した値を8で割ったあまりが7だったことを示す。

規　　格――*XC*

_iocs_defchr

用　途——外字パターンを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_defchr (int type, int code, const void *buff);
```

解　説——_iocs_defchr 関数は，*code* で指定した外字に *buff* で指定したパターンデータを設定する。

type には設定する外字のドット数を 8 または 12 で指定するが，0 を指定した場合は 8 を指定したとみなす。また，*code* にはシフト JIS コードか，JIS 漢字コードの 0x7621 ～ 0x777E までを指定する。

type に 0 か 8 を指定した場合は 32 バイト，12 を指定した場合は 72 バイトの外字パターンが *buff* に設定され，それぞれ 16×16 ドット，24×24 ドットの外字パターンとなる。

_iocs_defchr 関数は IOCS コール 0x0F を発行することによって処理される。

戻り値——成功した場合は 0 を返し，*code* が外字コードでなかった場合は−1 を返す。

規　格——*XC*

_iocs_densns

用　途——電卓処理を行う。

書　式——#include <sys/iocs.h>

```
void _iocs_densns (void);
```

解　説——_iocs_densns 関数は電卓処理を行う。

電卓処理は_iocs_b_keyinp 関数や_iocs_b_keysns 関数などのキー入力関数で処理される。

したがって，キーボードを使用しないでマウスだけを使用する場合は，ソフトキーボードが表示されていても電卓が使えないことがある。このような場合，_iocs_densns 関数を呼び出しながらマウス処理を行うことで電卓が使えるようになる。

_iocs_densns 関数は IOCS コール 0x7E を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_iocs_b_keyinp，_iocs_b_keysns

_iocs_dmamode

用　途——DMA の実行モードを調べる。

書　式——#include <sys/iocs.h>

```
int _iocs_dmamode (void);
```

解　説——_iocs_dmamode 関数は DMA の実行モードを調べる。

_iocs_dmamode 関数は IOCS コール 0x8D を発行することによって処理される。

戻り値——実行モードを返す。実行モードは次のとおり。

- 0x00　何もしていない
- 0x8A　転送中 (_iocs_dmamove 関数実行中)
- 0x8B　転送中 (_iocs_dmamov_l 関数実行中)
- 0x8C　転送中 (_iocs_dmamov_a 関数実行中)

規　格——*XC*

関連項目——_iocs_dmamov_a, _iocs_dmamov_l, _iocs_dmamove

_iocs_dmamov_a

用　　途——アレイチェインモードでDMA転送を行う。

書　　式——

```
#include <sys/iocs.h>
void _iocs_dmamov_a (const struct _chain *tbl,
                    void *addr, int mode, int cnt);
```

解　　説——_iocs_dmamov_a 関数はアレイチェインモードでDMA転送を行う。

*tbl*には転送元データチェインテーブルのアドレスを指定する。*tbl*に指定する _chain 構造体は次のとおり。

```
struct _chain {
    void *addr;          /* データバッファの先頭アドレス */
    unsigned short len; /* データバッファの長さ */
};
```

*addr*には転送先アドレスを指定し，*cnt*には転送データチェインテーブルの個数を指定する。

*mode*に指定する値の内容は次のとおり。

ビット7	転送方向 0 ... *tbl*から *addr*へ転送 1 ... *addr*から *tbl*へ転送
ビット6～4	つねに0
ビット3～2	MAC (addr1) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可
ビット1～0	DAC (addr2) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可

_iocs_dmamov_a 関数はIOCSコール 0x8B を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_iocs_dmamov_l

D

用　途——リンクアレイチェインモードでDMA転送を行う。

書　式——#include <sys/iocs.h>

```
void _iocs_dmamov_l (const struct _chain2 *tbl,
                     void *addr, int mode);
```

解　説——_iocs_dmamov_l 関数はリンクアレイチェインモードでDMA転送を行う。

*tbl*には転送元データチェインテーブルのアドレスを指定する。*tbl*に指定する_chain2構造体は次のとおり。

```
struct _chain2 {
    void *addr;                  /* データバッファの先頭アドレス */
    unsigned short len;          /* データバッファの長さ */
    const struct _chain2 *next;  /* 次のテーブルアドレス */
};
```

最後のテーブルのメンバnextには0を指定し，*addr*には転送先アドレスを指定する。

*mode*に指定する値の内容は次のとおり。

ビット7	転送方向 0 ... *tbl*から*addr*へ転送 1 ... *addr*から*tbl*へ転送
ビット6〜4	つねに0
ビット3〜2	MAC (addr1) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可
ビット1〜0	DAC (addr2) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可

_iocs_dmamov_l 関数はIOCSコール0x8Cを発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_dmamove

用　　途——DMA転送を行う。

書　　式——
```
#include <sys/iocs.h>
void _iocs_dmamove (void *addr1,
                    void *addr2, int mode, int len);
```

解　　説——_iocs_dmamove関数はDMA転送を行う。

addr1, *addr2*には転送の対象となるアドレスを指定し, *len*には転送バイト数を指定する。転送するバイト数は0xFF00バイト以下を指定すること。

*mode*に指定する値の内容は次のとおり。

ビット7	転送方向 0 ... *addr1*から*addr2*へ転送 1 ... *addr2*から*addr1*へ転送
ビット6～4	つねに0
ビット3～2	MAC (addr1) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可
ビット1～0	DAC (addr2) 00 ... カウントしない 01 ... インクリメント 10 ... デクリメント 11 ... 指定不可

_iocs_dmamove関数はIOCSコール0x8Aを発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_iocs_fill

用　途──グラフィック画面に塗りつぶしたボックスを描画する。

書　式──#include <sys/iocs.h>

```
int _iocs_fill (const struct _fillptr *ptr);
```

解　説──_iocs_fill 関数はグラフィック画面に塗りつぶしたボックスを描画する。*ptr* に指定する _fillptr 構造体は次のとおり。

```
struct _fillptr {
    short x1;             /* 始点の X 座標 */
    short y1;             /* 始点の Y 座標 */
    short x2;             /* 終点の X 座標 */
    short y2;             /* 終点の Y 座標 */
    unsigned short color; /* パレットコード */
};
```

_iocs_fill 関数は IOCS コール 0xBA を発行することによって処理される。

戻り値──成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格──*XC*

_iocs_fntget

用　途——漢字パターンを取得する。

書　式——#include <sys/iocs.h>

```
int _iocs_fntget (int type, int code, struct _fntbuf *fntbuf);
```

解　説——_iocs_fntget 関数は，*type* と *code* で指定した漢字パターンを *fntbuf* で指定した _fntbuf 構造体へ格納する。*fntbuf* のアドレスにはスーパーバイザ領域も指定可能なので，誤ったアドレスを与えないように注意すること。*fntbuf* に指定する _fntbuf 構造体は次のとおり。

```
struct _fntbuf {
    short xl;                /* 処理する X 方向のドット数 */
    short yl;                /* 処理する Y 方向のドット数 */
    unsigned char buffer[72]; /* 読み込みバッファ */
};
```

type には 8 または 12 を指定するが，0 を指定した場合は 8 とみなされる。また，*code* にはシフト JIS コードか，JIS 漢字コードの 0x2121 ～ 0x7E7E まで指定する。

type に 0 か 8 を指定した場合は 36 バイト，12 を指定した場合は 76 バイトの外字パターンが *fntbuf* に設定され，それぞれ 16 × 16 ドット，24 × 24 ドットの外字パターンとなる。また *fntbuf* に設定される内容は漢字パターンの縦，横それぞれのドット数およびパターンデータである。

_iocs_fntget 関数は IOCS コール 0x19 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_g_clr_on

用　途――グラフィック画面のクリア/表示を行う。

書　式――#include <sys/iocs.h>

```
void _iocs_g_clr_on (void);
```

解　説――_iocs_g_clr_on 関数は，グラフィック画面のクリアと表示モードの設定を行う。このときグラフィックパレットは初期状態に戻される。

_iocs_g_clr_on 関数は IOCS コール 0x90 を発行することによって処理される。

戻り値――なし。

規　格――*XC*

_iocs_getgrm

用　途——グラフィック画面からデータを読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_getgrm (struct _getptr *ptr);
```

解　説——_iocs_getgrm関数はグラフィック画面の読み込みを行う。*ptr*に指定する_getptr構造体は次のとおり。

```
struct _getptr {
    short x1;         /* 開始位置のX座標 */
    short y1;         /* 開始位置のY座標 */
    short x2;         /* 終了位置のX座標 */
    short y2;         /* 終了位置のY座標 */
    void *buf_start; /* バッファの開始アドレス */
    void *buf_end;   /* バッファの終了アドレス */
};
```

バッファにはパックされた形で，データが格納される。

_iocs_getgrm関数はIOCSコール0xBEを発行することによって処理される。

戻り値——成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格——*XC*

_iocs_gpalet

用　途——グラフィックパレットを設定する。

書　式——

```
#include <sys/iocs.h>
int _iocs_gpalet (int no, int color);
```

解　説——_iocs_gpalet 関数はグラフィックパレットを設定する。

no にはパレット番号 (0 ～ 15)/(0 ～ 255)/(0 ～ 65535) を指定し，*color* にはカラーコード (0 ～ 65535) を指定する。ただし，*color* に−1 を指定すると *no* で指定したパレットの読み込みを行う。

_iocs_gpalet 関数は IOCS コール 0x94 を発行することによって処理される。

戻り値——*color* に−1 を指定したときだけ，現在のカラーコードを返す。

規　格——*XC*

_iocs_hanjob

用　途――半濁点処理を行う。

書　式――#include <sys/iocs.h>

```
int _iocs_hanjob (char *end);
```

解　説――_iocs_hanjob 関数は半濁点処理を行う。

end には文字列の終端の null 文字のアドレスを指定する。最後の全角文字に半濁点処理ができないときは文字列に‘゜’を追加するため，null 文字の後に3バイト分の領域が必要である。また，‘゜’を追加した場合は‘゜’の後に null 文字を付加するので，文字列の途中を指定すると‘゜’の後の文字列が切れてしまう。したがって，文字列の途中は指定しないこと。

_iocs_hanjob 関数は IOCS コール 0xA5 を発行することによって処理される。

戻り値――変換ステータスを返す。変換ステータスは次のとおり。

- 0　最後の全角文字に半濁点処理をした
- 2　‘゜’を付加した

規　格――*XC*

関連項目――_iocs_dakjob

_iocs_home

用　途――グラフィック画面の表示開始位置を設定する。

書　式――

```
#include <sys/iocs.h>
int _iocs_home (int page, int x, int y);
```

解　説――_iocs_home 関数はグラフィック画面の表示開始位置を設定する。

*page*には，設定するページをビット 0 ～ 3 で指定する (ページ 3 ならビット 3)。すべてのビットが 0 ならば，現在のモードで有効なすべてのページが設定される。また，*x*，*y*には表示開始座標を指定する (0 ～ 1023)。

_iocs_home 関数は IOCS コール 0xB3 を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない
- −2　規定外の引数を指定した
- −3　現在の画面モードでは設定できない

規　格――*XC*

H

_iocs_hsvtorgb

用　途——HSV方式からRGB方式への色変換を行う。

書　式——
```
#include <sys/iocs.h>
int _iocs_hsvtorgb (int h, int s, int v);
```

解　説——_iocs_hsvtorgb関数は，*h*，*s*，*v*で指定したHSV形式の色コードをRGB形式のデータに変換する。

*h*は色の指定(色相)でそれぞれ32段階ごとのグループに分かれており，色合いも指定できる。指定できる値と色合いの対応は次のとおり。

- 0 ～ 31　赤から黄色
- 32 ～ 63　黄色から緑
- 64 ～ 95　緑からシアン
- 96 ～ 127　シアンから青
- 128 ～ 159　青からマゼンダ
- 160 ～ 191　マゼンダから赤
- 192 ～　禁止

*s*は白レベルの指定(飽和度)で，0で白，0x1Fで原色を表す。また，*v*は黒レベルの指定(明るさ)で，0で黒，0x1Fで原色を表す。

_iocs_hsvtorgb関数はIOCSコール0x12を発行することによって処理される。

戻り値——変換されたRGBコードを次のようなビットフィールドで返す。

```
00000000 00000000 GGGGGRRR RRBBBBB0
```

赤，青，緑はそれぞれ5ビットで，最下位ビットは必ず0である。

規　格——*XC*

_iocs_hsyncst

用　途──H-SYNC(水平同期信号) 割り込みを設定する。

書　式──#include <sys/iocs.h>

```
int _iocs_hsyncst (const void *addr);
```

解　説──_iocs_hsyncst 関数は，H-SYNC(水平同期信号) による割り込みを設定する。*addr* には割り込み処理アドレスを指定するが，0 を指定することで割り込みを禁止できる。

_iocs_hsyncst 関数は IOCS コール 0x6E を発行することによって処理される。

戻り値──割り込みが設定された場合は 0 を返し，すでに使用している場合は 0 以外の値を返す。

規　格──*XC*

_iocs_init_prn

用　　途——プリンタポートの初期化を行う。

書　　式——
```
#include <sys/iocs.h>
int _iocs_init_prn (int line, int width);
```

解　　説——`_iocs_init_prn` 関数はプリンタポートを初期化する。

line には1ページの行数−1を設定し，*width* には1行の桁数−1を指定する。ただし，いずれも `0xFF` を指定することで制限を与えないようにすることができる。*line* と *width* は `_iocs_outprn` 関数のための指定である。

`_iocs_init_prn` 関数は IOCS コール `0x3C` を発行することによって処理される。

戻り値——プリンタへの出力が可能ならば `0x000020` を返し，不可ならば `0x000000` を返す。

規　　格——*XC*

_iocs_inp232c

用　途——RS-232C の受信バッファから 1 バイトデータ読み込む。

書　式——
```
#include <sys/iocs.h>
int _iocs_inp232c (void);
```

解　説——_iocs_inp232c 関数は RS-232C の受信バッファ内の受信データを 1 バイトデータ読み込む。

_iocs_inp232c 関数は IOCS コール 0x32 を発行することによって処理される。

戻り値——読み込んだバイトデータを返す。

規　格——*XC*

_iocs_iplerr

用　途——IPLエラー時の処理を行う。

書　式——#include <sys/iocs.h>

```
void _iocs_iplerr (void);
```

解　説——_iocs_iplerr関数は，IPLプログラムでディスクエラーなどのエラーが発生した場合に再起動させるために使用する。

_iocs_iplerr関数はIOCSコール0xFEを発行することによって処理される。

戻り値——なし。_iocs_iplerr関数は決して戻ってこない。

規　格——*XC*

_iocs_isns232c

用　途——RS-232C の受信バッファ内にデータがあるかどうかを調べる。

書　式——

```
#include <sys/iocs.h>
int _iocs_isns232c (void);
```

解　説——_iocs_isns232c 関数は RS-232C の受信バッファ内に受信データがあるかどうか調べる。

_iocs_isns232c 関数は IOCS コール 0x33 を発行することによって処理される。

戻り値——0x10000+受信データを返すが，戻り値が 0 の場合は，受信バッファ内にデータがないことを示す。

規　格——*XC*

_iocs_jissft

用　途——JIS漢字コードからシフトJIS漢字コードに変換する。

書　式——#include <sys/iocs.h>

```
int _iocs_jissft (int code);
```

解　説——_iocs_jissft関数は，*code*で指定したJIS漢字コードをシフトJIS漢字コードに変換する。

_iocs_jissft関数はIOCSコール0xA1を発行することによって処理される。

戻り値——変換したJIS漢字コードを返すが，上位16ビットが0xFFFFならばエラーが発生したことを示す。

規　格——*XC*

_iocs_joyget

用　途――ジョイスティックのデータを取得する。

書　式――

```
#include <sys/iocs.h>
int _iocs_joyget (int no);
```

解　説――_iocs_joyget 関数は *no* で指定した番号のジョイスティックの状態を調べる。*no* は，0 でジョイスティックの 1 番，1 で 2 番となる。

_iocs_joyget 関数は IOCS コール 0x3B を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

7	6	5	4	3	2	1	0
1	A	B	1	右	左	下	上

B，A はそれぞれジョイスティックのボタンの状態であり，各対応ビットが “1” のときに押されていることを示す。また右，左，下，上はそれぞれジョイスティックのレバーの状態であり，“0” でその方向に倒されていることを示す。

規　格――*XC*

_iocs_ledmod

用　途——LEDつきキーの設定を行う。

書　式——#include <sys/iocs.h>

```
void _iocs_ledmod (int code, int onoff);
```

解　説——_iocs_ledmod関数は*code*で指定したLEDつきキーを，*onoff*で指定したモードに設定する。*code*には0～6を指定し，*onoff*には1(ON)，0(OFF)を指定する。

*code*に指定する値は次のとおり。

6	5	4	3
全角	ひらがな	INS	CAPS

2	1	0
コード入力	ローマ字	かな

_iocs_ledmod関数はIOCSコール0x0Dを発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_line

用　途――グラフィック画面にラインを描画する。

書　式――#include <sys/iocs.h>

```
int _iocs_line (const struct _lineptr *ptr);
```

解　説――_iocs_line 関数はグラフィック画面にラインを描画する。*ptr* に指定する _lineptr 構造体は次のとおり。

```
struct _lineptr {
    short x1;                   /* 始点の X 座標 */
    short y1;                   /* 始点の Y 座標 */
    short x2;                   /* 終点の X 座標 */
    short y2;                   /* 終点の Y 座標 */
    unsigned short color;       /* パレットコード */
    unsigned short linestyle; /* ラインスタイル */
};
```

_iocs_line 関数は IOCS コール 0xB8 を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格――*XC*

L

_iocs_lof232c

用　途——RS-232C の受信バッファ内のデータ数を調べる。

書　式——

```
#include <sys/iocs.h>
int _iocs_lof232c (void);
```

解　説——_iocs_lof232c 関数は RS-232C の受信バッファ内のデータ数を調べる。

_iocs_lof232c 関数は IOCS コール 0x31 を発行することによって処理される。

戻り値——受信バッファ内のデータ数を返す。

規　格——*XC*

_iocs_ms_curgt

用　途――マウスカーソルの座標を調べる。

書　式――#include <sys/iocs.h>

```
int _iocs_ms_curgt (void);
```

解　説――_iocs_ms_curgt 関数はマウスカーソルの座標を調べる。

_iocs_ms_curgt 関数は IOCS コール 0x75 を発行することによって処理される。

戻り値――戻り値の上位 16 ビットに X 座標を返し，下位 16 ビットに Y 座標を返す。

規　格――*XC*

_iocs_ms_curof

用　途――マウスカーソルを消去する。

書　式――#include <sys/iocs.h>

```
void _iocs_ms_curof (void);
```

解　説――_iocs_ms_curof 関数はマウスカーソルを消去する。

_iocs_ms_curof 関数は IOCS コール 0x72 を発行することによって処理される。

戻り値――なし。

規　格――*XC*

_iocs_ms_curon

用　途——マウスカーソルを表示する。

書　式——#include <sys/iocs.h>

```
void _iocs_ms_curon (void);
```

解　説——_iocs_ms_curon 関数はマウスカーソルを表示する。

_iocs_ms_curon 関数は IOCS コール 0x71 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

M

_iocs_ms_curst

用　　途――マウスカーソルの座標を設定する。

書　　式――#include <sys/iocs.h>

```
int _iocs_ms_curst (int x, int y);
```

解　　説――_iocs_ms_curst 関数はマウスカーソルの座標を，x，yで指定した座標に設定する。

_iocs_ms_curst 関数は IOCS コール 0x76 を発行することによって処理される。

戻 り 値――正常に設定できた場合は 0 を返し，失敗した場合は−1 を返す。

規　　格――*XC*

_iocs_ms_getdt

用　途——マウスの移動量とボタンの状態を調べる。

書　式——
```
#include <sys/iocs.h>
int _iocs_ms_getdt (void);
```

解　説——_iocs_ms_getdt 関数はマウスの移動量とボタンの状態を調べる。

_iocs_ms_getdt 関数は IOCS コール `0x74` を発行することによって処理される。

戻り値——戻り値は次のとおり。

- ビット 31 ～ 24　X 方向移動量
- ビット 23 ～ 16　Y 方向移動量
- ビット 15 ～ 8　左ボタンの状態 (0xFF で ON，0 で OFF)
- ビット 7 ～ 0　右ボタンの状態 (0xFF で ON，0 で OFF)

規　格——*XC*

M

_iocs_ms_init

用　途——マウスを初期化する。

書　式——#include <sys/iocs.h>

```
void _iocs_ms_init (void);
```

解　説——_iocs_ms_init 関数はマウスの初期化を行う。

_iocs_ms_init 関数は IOCS コール 0x70 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_ms_limit

用　途——マウスカーソルの移動範囲を設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_ms_limit (int xs, int ys, int xe, int ye);
```

解　説——_iocs_ms_limit 関数はマウスカーソルの移動範囲を，$xs \leq x \leq xe$，$ys \leq y \leq ye$ に設定する。

_iocs_ms_limit 関数は IOCS コール 0x77 を発行することによって処理される。

戻り値——正常に設定できた場合は 0 を返し，失敗した場合は−1 を返す。

規　格——*XC*

_iocs_ms_offtm

用　途——マウスボタンを離すまでの時間を調べる。

書　式——
```
#include <sys/iocs.h>
int _iocs_ms_offtm (int mode, int max);
```

解　説——_iocs_ms_offtm 関数はマウスボタンを離すまでの時間を調べる。*mode* が 0 ならば左，−1 ならば右のボタンを調べる。*max* には待ち時間の最大値を指定するが，0 を指定するといつまでも待ち続ける。

_iocs_ms_offtm 関数は IOCS コール 0x78 を発行することによって処理される。

戻り値——待ち時間の残り (1 ～ 65534) を返す。ただし，0 の場合はドラッグが行われたことを示し，−1 の場合は待ち時間の最大値を越えたことを示す。

規　格——*XC*

_iocs_ms_ontm

用　途——マウスボタンを押すまでの時間を調べる。

書　式——

```
#include <sys/iocs.h>
int _iocs_ms_ontm (int mode, int max);
```

解　説——_iocs_ms_ontm 関数はマウスボタンを押すまでの時間を調べる。*mode* が 0 ならば左，−1 ならば右のボタンを調べる。*max* には待ち時間の最大値を指定するが，0 を指定するといつまでも待ち続ける。

_iocs_ms_ontm 関数は IOCS コール 0x79 を発行することによって処理される。

戻り値——待ち時間の残り (1 ～ 65534) を返す。ただし，0 の場合はドラッグが行われたことを示し，−1 の場合は待ち時間の最大値を越えたことを示す。

規　格——*XC*

M

_iocs_ms_patst

用　途——マウスカーソルのパターンを定義する。

書　式——#include <sys/iocs.h>

```
void _iocs_ms_patst (int no, const struct _patst *addr);
```

解　説——_iocs_ms_patst 関数は，*no*で指定したマウスカーソルのパターンを設定する。

*addr*にはパターンデータのアドレスを指定する。*addr*に指定する_patst 構造体は次のとおり。

```
struct _patst {
    short offsetx;      /* パターン左端からマウス座標までのドット数 */
    short offsety;      /* パターン上端からマウス座標までのドット数 */
    short shadow[16];  /* パターンの陰のデータ */
    short pattern[16]; /* パターンの表示データ */
};
```

_iocs_ms_patst 関数は IOCS コール 0x7A を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_ms_sel

用　途──マウスカーソルを選択する。

書　式──#include <sys/iocs.h>

```
void _iocs_ms_sel (int no);
```

解　説──_iocs_ms_sel 関数は，*no* で指定したマウスカーソルを選択する。

_iocs_ms_sel 関数は IOCS コール 0x7B を発行することによって処理される。

戻り値──なし。

規　格──*XC*

M

_iocs_ms_sel2

用　　途――マウスカーソルを複数選択し，アニメーションを作成する。

書　　式――
```
#include <sys/iocs.h>
void _iocs_ms_sel2 (const short *tbl);
```

解　　説――_iocs_ms_sel2 関数はマウスカーソルを複数選択して，アニメーションを作成する。*tbl*にはカーソル番号テーブルのアドレスを指定する。カーソル番号テーブルは short 型の配列で，最終データには−1 を指定すること。

_iocs_ms_sel2 関数は IOCS コール 0x7C を発行することによって処理される。

戻 り 値――なし。

規　　格――*XC*

_iocs_ms_stat

用　　途——マウスカーソルの表示モードを調べる。

書　　式——#include <sys/iocs.h>

```
int _iocs_ms_stat (void);
```

解　　説——_iocs_ms_stat 関数はマウスカーソルの表示モードを調べる。

_iocs_ms_stat 関数は IOCS コール 0x73 を発行することによって処理される。

戻 り 値——マウスカーソルが表示中ならば 0xFFFF を返し，表示中でなければ 0 を返す。

規　　格——*XC*

M

_iocs_ontime

用　途――電源投入またはリセットしてからの経過時間を調べる。

書　式――#include <sys/iocs.h>

```
int _iocs_ontime (void);
```

解　説――_iocs_ontime 関数が電源投入されるか，またはリセットしてからの経過時間を調べる。

_iocs_ontime 関数は IOCS コール 0x7F を発行することによって処理される。

戻り値――0秒～23時間59分59秒99(0～8639999)までの値を返す。この値は1日を越えると0にリセットされる。

規　格――*XC*

_iocs_opmintst

用　　途――FM 音源 IC (YM2151) による割り込みを設定する。

書　　式――

```
#include <sys/iocs.h>
int _iocs_opmintst (const void *addr);
```

解　　説――_iocs_opmintst 関数は，FM 音源 (YM2151) による割り込みを設定する。*addr* には割り込み処理アドレスを指定するが，0 を指定することで割り込みを禁止できる。

なお，FM 音源デバイスドライバ (“OPMDRV.X” など) を組み込んでいる状態で割り込みの変更を行ってはならない。

_iocs_opmintst 関数は IOCS コール 0x6A を発行することによって処理される。

戻 り 値――割り込みが設定された場合は 0 を返し，すでに使用している場合は 0 以外の値を返す。

規　　格――*XC*

O

_iocs_opmset

用　　途――FM 音源 (YM2151) にデータを書き込む。

書　　式――#include <sys/iocs.h>

```
void _iocs_opmset (int addr, int data);
```

解　　説――_iocs_opmset 関数は FM 音源 (YM2151) にデータを書き込む。

addr には書き込むレジスタのアドレスを指定し，*data* には書き込むデータを指定する。

データの書き込みは FM 音源内部のビジーフラグの状態にしたがって行われる。ただし，FM 音源デバイスドライバ ("OPMDRV.X" など) が動作中でかつ曲を演奏している場合はデータの書き込みを行ってはならない。

_iocs_opmset 関数は IOCS コール 0x68 を発行することによって処理される。

戻 り 値――なし。

規　　格――*XC*

_iocs_opmsns

用　途──FM 音源 (YM2151) のステータスを読む。

書　式──#include <sys/iocs.h>

```
int _iocs_opmsns (void);
```

解　説──_iocs_opmsns 関数は FM 音源 (YM2151) のステータスを読み込む。

_iocs_opmsns 関数は IOCS コール 0x69 を発行することによって処理される。

戻り値──FM 音源 (YM2151) のステータスを返す。ステータスのビットフィールドの内容は次のとおり。

- ビット 7　書き込み禁止フラグ (0 で書き込み可)
- ビット 1　タイマ A オーバフロー
- ビット 0　タイマ B オーバフロー

規　格──*XC*

_iocs_os_curof

用　　途——カーソルを消去する。

書　　式——#include <sys/iocs.h>

```
#include <sys/iocs.h>
void _iocs_os_curof (void);
```

解　　説——_iocs_os_curof 関数はカーソルを消去する（"esc[>5h" を出力した場合と同じ）。

_iocs_os_curof 関数はIOCSコール 0xAF を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_iocs_os_curon

用　途——カーソルを表示する。

書　式——#include <sys/iocs.h>

```
void _iocs_os_curon (void);
```

解　説——_iocs_os_curon 関数はカーソルを表示する（“esc[>5l” を出力した場合と同じ）。

_iocs_os_curon 関数は IOCS コール 0xAE を発行することによって処理される。

戻り値——なし。

規　格——*XC*

O

_iocs_osns232c

用　途——RS-232C が送信可能かどうかを調べる。

書　式——

```
#include <sys/iocs.h>
int _iocs_osns232c (void);
```

解　説——_iocs_osns232c 関数は RS-232C が送信可能な状態かどうかを調べる。

_iocs_osns232c 関数は IOCS コール 0x34 を発行することによって処理される。

戻り値——送信可能 (バッファが空で，フロー制御も送信可能の状態) ならば 0 以外の値を返し，送信不可能ならば 0 を返す。

規　格——*XC*

_iocs_out232c

用　途——RS-232C へ 1 バイト送信する。

書　式——#include <sys/iocs.h>

```
void _iocs_out232c (int code);
```

解　説——_iocs_out232c 関数は，*code* で指定した 1 バイトデータを RS-232C へ送信する。

_iocs_out232c 関数は IOCS コール 0x34 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

O

_iocs_outlpt

用　途——プリンタへの直接出力を行う。

書　式——
```
#include <sys/iocs.h>
void _iocs_outlpt (int code);
```

解　説——_iocs_outlpt 関数は *code* で指定した1バイトデータを，プリンタへ直接出力する (漢字処理などは行わない)。

_iocs_outlpt 関数は IOCS コール 0x3E を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_outprn

用　途――プリンタにデータを出力する。

書　式――#include <sys/iocs.h>

```
void _iocs_outprn (int code);
```

解　説――_iocs_outprn 関数は，*code* で指定した 1 バイトのデータを，プリンタへ出力する (シフト JIS コードで漢字処理を行う)。

_iocs_outprn 関数は IOCS コール 0x3F を発行することによって処理される。

戻り値――なし。

規　格――*XC*

_iocs_paint

用　途――グラフィック画面のペイントを行う。

書　式――#include <sys/iocs.h>

```
int _iocs_paint (struct _paintptr *ptr);
```

解　説――_iocs_paint 関数はグラフィック画面のペイントを行う。*ptr* に指定する _paintptr 構造体は次のとおり。

```
struct _paintptr {
    short x;                /* ペイント開始位置の X 座標 */
    short y;                /* ペイント開始位置の Y 座標 */
    unsigned short color;   /* パレットコード */
    void *buf_start;        /* 作業領域先頭アドレス */
    void *buf_end;          /* 作業領域終了アドレス */
};
```

_iocs_paint 関数は IOCS コール 0xBC を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格――*XC*

_iocs_point

用　途――グラフィック画面の指定座標のパレットコードを調べる。

書　式――#include <sys/iocs.h>

```
int _iocs_point (const struct _pointptr *ptr);
```

解　説――_iocs_point 関数は，指定したグラフィック画面の座標のパレットコードを調べる。*ptr* に指定する _pointptr 構造体は次のとおり。

```
struct _pointptr {
    short x;              /* X 座標 */
    short y;              /* Y 座標 */
    unsigned short color; /* パレットコード */
};
```

_iocs_point 関数は IOCS コール 0xB7 を発行することによって処理される。

戻り値――成功した場合は 0 を返し，パレットコードが _pointptr 構造体の color に格納される。失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格――*XC*

P

_iocs_prnintst

用　　途――プリンタ割り込みを設定する。

書　　式――#include <sys/iocs.h>

```
int _iocs_prnintst (const void *addr);
```

解　　説――_iocs_prnintst 関数はプリンタ割り込みを設定する。*addr* には割り込み処理アドレスを指定するが，0を指定することで割り込みを禁止できる。

_iocs_prnintst 関数は IOCS コール 0x6F を発行することによって処理される。

戻 り 値――割り込みが設定された場合は0を返し，すでに使用している場合は0以外の値を返す。

規　　格――*XC*

_iocs_pset

用　　途——グラフィック画面に点を描画する。

書　　式——#include <sys/iocs.h>

```
int _iocs_pset (const struct _psetptr *ptr);
```

解　　説——_iocs_pset 関数はグラフィック画面の指定した座標に点を描く。*ptr* に指定する _psetptr 構造体は次のとおり。

```
struct _psetptr {
    short x;              /* 描画する X 座標 */
    short y;              /* 描画する Y 座標 */
    unsigned short color; /* 描画パレットコード */
};
```

_iocs_pset 関数は IOCS コール 0xB6 を発行することによって処理される。

戻 り 値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　　格——*XC*

P

_iocs_putgrm

用　途——グラフィック画面にデータを書き込む。

書　式——

```
#include <sys/iocs.h>
int _iocs_putgrm (const struct _putptr *ptr);
```

解　説——_iocs_putgrm関数はグラフィック画面にデータを書き込む。*ptr*に指定する_putptr構造体は次のとおり。

```
struct _putptr {
    short x1;               /* 開始位置の X 座標 */
    short y1;               /* 開始位置の Y 座標 */
    short x2;               /* 終了位置の X 座標 */
    short y2;               /* 終了位置の Y 座標 */
    const void *buf_start; /* バッファの開始アドレス */
    const void *buf_end;   /* バッファの終了アドレス */
};
```

バッファにはパックされた形で，データを格納すること。

_iocs_putgrm関数はIOCSコール0xBFを発行することによって処理される。

戻り値——成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格——*XC*

_iocs_rmacnv

用　途――ローマ字/かな変換を行う。

書　式――#include <sys/iocs.h>

```
int _iocs_rmacnv (int code, char *wptr, char *aptr);
```

解　説――_iocs_rmacnv 関数はローマ字変換を行う。*code*には文字コードを指定し，*wptr*にはワークポインタを，*aptr*には変換結果を格納する領域へのポインタを指定する。

_iocs_rmacnv 関数はIOCS コール 0xA3 を発行することによって処理される。

戻り値――変換ステータスを返す。変換ステータスは次のとおり。

- 0　変換の途中
- −1　変換不可能
- 上記以外　変換文字数を示す (*wptr*の領域に変換途中の文字が残っている可能性がある)

規　格――*XC*

_iocs_romver

用　途——ROMのバージョンと作成日付を調べる。

書　式——
```
#include <sys/iocs.h>
int _iocs_romver (void);
```

解　説——_iocs_romver 関数はROMのバージョンと作成日付を調べる。

_iocs_romver 関数はIOCSコール 0x8F を発行することによって処理される。

戻り値——ROMのバージョン情報を次のようなビットフィールドで返す。

```
VVVVVVVV YYYYYYYY MMMMMMMM DDDDDDDD
```

- VVVVVVVV　バージョン
- YYYYYYYY　年
- MMMMMMMM　月
- DDDDDDDD　日

規　格——*XC*

_iocs_scroll

用　途――テキスト/グラフィックの表示開始座標を設定する。

書　式――
```
#include <sys/iocs.h>
void _iocs_scroll (int mode, int x, int y);
```

解　説――_iocs_scroll 関数は *mode* で指定した画面の表示開始座標を，*x*，*y* で指定した座標に設定する。

mode に指定できる値は次のとおり。

- 0　グラフィックスクリーン 0 の設定
- 1　グラフィックスクリーン 1 の設定
- 2　グラフィックスクリーン 2 の設定
- 3　グラフィックスクリーン 3 の設定
- 8　テキスト画面の設定

_iocs_scroll 関数は IOCS コール 0x1D を発行することによって処理される。

戻り値――なし。

規　格――*XC*

S

_iocs_set232c

用　途——RS–232C の通信モードを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_set232c (int mode);
```

解　説——_iocs_set232c 関数は，RS-232C の通信モードを *mode* で指定したモードに設定する。*mode* に指定するビットフィールドの内容は次のとおり。

SS PP LL X 0 0000 BBBB

上記形式の SS でストップビットの指定をし，次の値をとる。

- 01　1 ビット
- 10　1.5 ビット
- 11　2 ビット
- 00　2 ビット

上記形式の PP でパリティビットの指定をし，次の値をとる。

- 01　奇数
- 11　偶数
- 00　パリティなし
- 10　パリティなし

上記形式の LL でビット長の指定をし，次の値をとる。

- 00　5 ビット以下
- 01　6 ビット
- 10　7 ビット
- 11　8 ビット

上記形式の X で X フロー制御の指定をし，次の値をとる。

- 0　処理しない
- 1　処理する

上記形式の BBBB で通信速度の指定をし，次の値をとる。

- 0000　75bps
- 0001　150bps

- 0010　300bps
- 0011　600bps
- 0100　1200bps
- 0101　2400bps
- 0110　4800bps
- 0111　9600bps
- 1000　19200bps
- 1001　38400bps

19200bps，38400bps の通信速度での処理は，RS-232C ドライバが処理する。ただし，使用するドライバによっては 19200bps や 38400bps の指定ができない場合がある。またドライバを組み込んでいない場合，19200bps，38400bps はそれぞれ 75bps，150bps と認識される。

_iocs_set232c 関数は IOCS コール 0x30 を発行することによって処理される。

戻り値——*mode* に−1 を指定したときだけ，現在のモードを返す。

規　格——*XC*

_iocs_sftjis

用　途——シフトJIS漢字コードからJIS漢字コードに変換する。

書　式——#include <sys/iocs.h>

```
int _iocs_sftjis (int code);
```

解　説——_iocs_sftjis 関数は *code* で指定したシフト JIS 漢字コードを，JIS 漢字コードに変換する。

_iocs_sftjis 関数は IOCS コール 0xA0 を発行することによって処理される。

戻り値——変換した JIS 漢字コードを返すが，上位 16 ビットが 0xFFFF ならばエラーが発生したことを示す。

規　格——*XC*

_iocs_skey_mod

用　途——ソフトキーボードを制御する。

書　式——
```
#include <sys/iocs.h>
int _iocs_skey_mod (int mode, int x, int y);
```

解　説——_iocs_skey_mod 関数はソフトキーボードの制御を行う。

*mode*には次の値を指定する。

- 0　ソフトキーボード消去
- 1　ソフトキーボード表示 (表示状態では無効)
- 2　ソフトキーボード表示状態の検査
- −1　ソフトキーボード自動制御 (右ボタンで表示/消去)

*mode*でソフトキーボードの表示を選択した場合，*x*，*y*にはソフトキーボードを表示する座標を指定する。

_iocs_skey_mod 関数は IOCS コール 0x7D を発行することによって処理される。

戻り値——ソフトキーボードの状態を返す。

- 0　消去状態
- 1　表示状態

規　格——*XC*

S

_iocs_skeyset

用　　途——指定したキーが押されたことにする。

書　　式——#include <sys/iocs.h>

```
int _iocs_skeyset (int code);
```

解　　説——_iocs_skeyset関数は，*code*で指定したキーがキーボードから入力されたかのように，内部情報を設定する。

キーの番号は，_iocs_bitsns関数で指定するキーコードグループを8倍したものに，対象キーのビット位置を加えた値を指定する。たとえば，'A'が入力されたかのようにするには，'A'のグループが3，ビット位置が6であることから，3×8+6=0x1Eを設定すればよいことになる。

_iocs_skeyset関数はIOCSコール0x04を発行することによって処理される。

戻り値——指定したキーコードグループの押下げ状態を返す。該当ビットが1ならば，そのキーが押されていることを示す。

規　　格——*XC*

関連項目——_iocs_bitsns

_iocs_snsprn

用　途——プリンタ出力が可能かどうかを調べる。

書　式——
```
#include <sys/iocs.h>
int _iocs_snsprn (void);
```

解　説——_iocs_snsprn 関数はプリンタへの出力が可能かどうかを調べる。

_iocs_snsprn 関数は IOCS コール 0x3D を発行することによって処理される。

戻り値——プリンタへの出力が可能ならば 0 以外を返し，出力不可ならば 0 を返す。

規　格——*XC*

_iocs_sp_cgclr

用　途――PCGをクリアする。

書　式――#include <sys/iocs.h>

```
int _iocs_sp_cgclr (int code);
```

解　説――_iocs_sp_cgclr関数は，*code*で指定したPCGをクリアする。

_iocs_sp_cgclr関数はIOCSコール0xC3を発行することによって処理される。

戻り値――成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー(画面サイズが768 × 512のときなど)

規　格――*XC*

_iocs_sp_defcg

用　途──PCG を設定する。

書　式──#include <sys/iocs.h>

```
int _iocs_sp_defcg (int code, int size, const void *addr);
```

解　説──_iocs_sp_defcg 関数は *code* で指定した PCG を *addr* で指定したパターンデータで設定する。

code にはパターンコード (0 ～ 255) を指定し，*size* にはパターンサイズを指定する。

size 指定できる値は次のとおり。

- 0　8 × 8 ドットパターン
- 1　16 × 16 ドットパターン

また，*addr* にはパターンデータバッファのアドレスを指定する。このときパターンデータのサイズは *size* により次のようになる。

- 0　32 バイト
- 1　128 バイト

_iocs_sp_defcg 関数は IOCS コール 0xC4 を発行することによって処理される。

戻り値──成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格──*XC*

S

_iocs_sp_gtpcg

用　途——PCGを読み込む。

書　式——

```
#include <sys/iocs.h>
int _iocs_sp_gtpcg (int code, int size, void *addr);
```

解　説——_iocs_sp_gtpcg関数は*code*で指定したPCGを読み込み，*addr*で指定した領域に格納する。

*code*にはパターンコード(0～255)を指定し，*size*にはパターンサイズを指定する。

*size*に指定できる値は次のとおり。

- 0　8×8ドットパターン
- 1　16×16ドットパターン

また，*addr*にはパターンデータバッファのアドレスを指定する。このときパターンデータのサイズは*size*により次のようになる。

- 0　32バイト
- 1　128バイト

_iocs_sp_gtpcg関数はIOCSコール0xC5を発行することによって処理される。

戻り値——成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー(画面サイズが768×512のときなど)

規　格——*XC*

_iocs_sp_init

用　途——スプライト画面を初期化する。

書　式——#include <sys/iocs.h>

```
int _iocs_sp_init (void);
```

解　説——_iocs_sp_init 関数はスプライト画面の初期化を行う。

_iocs_sp_init 関数は IOCS コール 0xC0 を発行することによって処理される。

戻り値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格——*XC*

_iocs_sp_off

用　　途――スプライト画面を非表示にする。

書　　式――#include <sys/iocs.h>

```
void _iocs_sp_off (void);
```

解　　説――_iocs_sp_off 関数はスプライト画面が表示されないようにする。

_iocs_sp_off 関数は IOCS コール 0xC2 を発行することによって処理される。

戻り値――なし。

規　　格――***XC***

_iocs_sp_on

用　　途──スプライト画面を表示する。

書　　式──
```
#include <sys/iocs.h>
int _iocs_sp_on (void);
```

解　　説──_iocs_sp_on 関数は**スプライト画面**が表示されるようにする。

_iocs_sp_on 関数は IOCS コール 0xC1 を発行することによって処理される。

戻 り 値──成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　　格──*XC*

_iocs_sp_reggt

用　途――スプライトレジスタを読み込む。

書　式――#include <sys/iocs.h>

```
int _iocs_sp_reggt (int no, int *x, int *y, int *code, int *prw);
```

解　説――_iocs_sp_reggt 関数はスプライトレジスタの読み込みを行う。

*no*にはスプライトプレーン番号 (0 ～ 127) を指定し，*x*，*y*にはそれぞれ X 座標，Y 座標を格納する領域へのポインタを指定する。返される値は 0 ～ 1023 の範囲である。

また，*prw*にはプライオリティを格納する領域へのポインタを指定する。返される値は 0 ～ 3 の範囲であり意味は次のとおり。

- 0　スプライトは表示しない
- 1　スプライトは BG0，BG1 よりも後ろ
- 2　スプライトは BG0 と BG1 の間
- 3　スプライトは BG0，BG1 よりも前

*code*には次の内容を格納する領域へのポインタを指定する。

- ビット 16 ～ 32　つねに 0
- ビット 15　垂直反転 (0=標準，1=反転)
- ビット 14　水平反転 (0=標準，1=反転)
- ビット 12 ～ 13　つねに 0
- ビット 8 ～ 11　パレットブロック番号 (0 ～ 15)
- ビット 0 ～ 7　パターンコード (0 ～ 255)

_iocs_sp_reggt 関数は IOCS コール 0xC7 を発行することによって処理される。

戻り値――成功した場合は 0 を返し，X 座標，Y 座標，パターンコード，プライオリティがそれぞれ指定した領域に格納される。失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格――*XC*

_iocs_sp_regst

用　途——スプライトレジスタを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_sp_regst (int no, int mode,
                    int x, int y, int code, int prw);
```

解　説——_iocs_sp_regst 関数はスプライトレジスタの設定を行う。

no にはスプライトプレーン番号 (0 ～ 127) を指定し，*mode* には設定に先だって**垂直帰線期間**を検出するかどうかを設定する。最上位ビットが 0 ならばレジスタは垂直帰線期間を検出後に設定され，1 ならば無条件に設定される。

x，*y* にはそれぞれ X 座標，Y 座標 (0 ～ 1023) を，*code* にはパターンコードを，*prw* にはスプライトの**プライオリティ**(0 ～ 3) を設定する。いずれも−1 が指定された場合は設定を変更せずに現在の設定値を使用する。詳細は_iocs_sp_reggt 関数を参照のこと。

_iocs_sp_regst 関数は IOCS コール 0xC6 を発行することによって処理される。

戻り値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　画面モードエラー (画面サイズが 768 × 512 のときなど)

規　格——*XC*

関連項目——_iocs_sp_reggt

変　更——従来では，引数 *mode* は *no*，*mode* の 2 つの引数にわかれていたが，元々の IOCS コールの仕様通り，現在の *LIBC* ではこれらを 1 つにまとめた。

［新書式］

```
int _iocs_sp_regst (int mode, int x, int y, int code, int prw);
```

_iocs_spalet

用　途——スプライトパレットの設定/読み込みを行う。

書　式——#include <sys/iocs.h>

```
int _iocs_spalet (int mode, int block, int color);
```

解　説——_iocs_spalet 関数はスプライトパレットの設定と読み込みを行う。

*mode*には，垂直帰線期間と同期してパレットを変更するか，または同期を取らずに変更するかを指定する。垂直帰線期間を検出しないでパレットコードを変更すると，画面にちらつきが発生するので注意すること。

*mode*に設定する値は次のとおり。

- 0x00 ～ 0x0F　垂直帰線期間を検出後に設定
- 0x80 ～ 0x8F　検出しない

*block*にはパレットブロック (0 ～ 15) を指定し，*color*にはカラーコード (0 ～ 65535) を指定する。ただし，カラーコードに−1 を指定した場合，カラーコードは変更せず，読み込みだけを行う。

_iocs_spalet 関数は IOCS コール 0xCF を発行することによって処理される。

戻り値——成功した場合は設定前のカラーコードを返し，失敗した場合は次の値を返す。

- −2　パラメータエラー (パレットブロック 0 のアクセス)

規　格——*XC*

関連項目——_iocs_bgtextcl

_iocs_symbol

用　途――グラフィック画面に文字を描画する。

書　式――
```
#include <sys/iocs.h>
int _iocs_symbol (const struct _symbolptr *ptr);
```

解　説――_iocs_symbol 関数はグラフィック画面に文字を描画する。*ptr*に指定する _symbolptr 構造体は次のとおり。

```
struct _symbolptr {
    short x1;                       /* 描画を開始する X 座標 */
    short y1;                       /* 描画を開始する Y 座標 */
    const char *string_address;     /* 描画文字列のアドレス */
    unsigned char mag_x;            /* 横倍率 */
    unsigned char mag_y;            /* 縦倍率 */
    unsigned short color;           /* パレットコード */
    unsigned char font_type;        /* 文字フォントのタイプ */
    unsigned char angle;            /* 角度コード */
};
```

x1，y1 にはグラフィック画面の描画開始座標を指定する。通常，座標は描画する文字列の左上隅の座標を指定するが，回転角度が 90 度，180 度，270 度の場合はそれぞれ左下，右下，右上隅の座標となる。

mag_x には文字の横倍率を，mag_y には文字の縦倍率をそれぞれ指定する。ただし，この倍率の指定は文字の縦横に対しての倍率であり，画面の縦横に対してではないので注意すること。

また color には表示色のパレットコードを，font_type には次の文字フォントのタイプを指定する。指定することのできるフォントタイプは次のとおり。

- 0　12 × 12 ドットフォント
- 1　16 × 16 ドットフォント
- 2　24 × 24 ドットフォント

angle には次の回転角コードを指定する。

- 0　通常
- 1　90 度回転
- 2　180 度回転
- 3　270 度回転

_iocs_symbol 関数は IOCS コール 0xBD を発行することによって処理される。

S

戻り値――成功した場合は0を返し，失敗した場合は次の値を返す。

- −1 グラフィックは使用できない

規　　格――*XC*

_iocs_tcolor

用　途──テキストのカラーを選択する。

書　式──#include <sys/iocs.h>

```
int _iocs_tcolor (int color);
```

解　説──_iocs_tcolor 関数はテキストカラーを選択する。

color には 0 ～ 15 までの値が指定可能であるが，原則としては 1，2，4，8 のみ指定すること。

- 1　0xE00000 ～ 0xE1FFFF のテキスト画面 (テキスト 0)
- 2　0xE20000 ～ 0xE3FFFF のテキスト画面 (テキスト 1)
- 4　0xE40000 ～ 0xE5FFFF のテキスト画面 (テキスト 2)
- 8　0xE60000 ～ 0xE7FFFF のテキスト画面 (テキスト 3)

_iocs_tcolor 関数は IOCS コール 0x15 を発行することによって処理される。

戻り値──なし。

規　格──*XC*

_iocs_textget

用　途――テキスト画面からデータを読み込む。

書　式――#include <sys/iocs.h>

```
void _iocs_textget (int x, int y, struct _fntbuf *buff);
```

解　説――_iocs_textget 関数は，*x* および *y* で指定したテキスト画面の任意の座標から，パターンデータを読み込み，結果を *buff* で指定したバッファ領域に格納する。

buff にはスーパーバイザ空間も指定できるので，誤ったアドレスを指定してはならない。また，*buff* に指定する _fntbuf 構造体には読み込む X 方向，Y 方向のドット数を short で指定する。

もし，_fntbuf 構造体に入らないサイズを指定する場合は，独自に malloc 関数などを使って領域を確保すること。*buff* に指定する _fntbuf 構造体は次のとおり。

```
struct _fntbuf {
    short xl;                  /* 処理する X 方向のドット数 */
    short yl;                  /* 処理する Y 方向のドット数 */
    unsigned char buffer[72]; /* 読み込みバッファ */
};
```

_iocs_textget 関数は IOCS コール 0x1A を発行することによって処理される。

戻り値――なし。

規　格――*XC*

関連項目――_iocs_textput，malloc

_iocs_textput

用　途――テキスト画面にデータを書き込む。

書　式――#include <sys/iocs.h>

```
void _iocs_textput (int x, int y, const struct _fntbuf *buff);
```

解　説――_iocs_textput 関数は，*x* および *y* で指定したテキスト画面の任意の座標から *buff* で指定したパターンデータを書き込む。

buff にはスーパーバイザ空間も指定できるので，誤ったアドレスを指定してはならない。また，*buff* に指定する _fntbuf 構造体には書き込む X 方向，Y 方向のドット数，およびパターンデータを short 型で指定する。

もし，_fntbuf 構造体に入らないサイズを指定する場合は，独自に malloc 関数などを使って領域を確保すること。*buff* に指定する _fntbuf 構造体は次のとおり。

```
struct _fntbuf {
    short xl;                    /* 処理する X 方向のドット数 */
    short yl;                    /* 処理する Y 方向のドット数 */
    unsigned char buffer[72]; /* 書き込みバッファ */
};
```

_iocs_textput 関数は IOCS コール 0x1B を発行することによって処理される。

戻り値――なし。

規　格――*XC*

関連項目――_iocs_textget，malloc

T

_iocs_tgusemd

用　途――画面モードを設定/取得する。

書　式――
```
#include <sys/iocs.h>
int _iocs_tgusemd (int text_gr, int mode);
```

解　説――_iocs_tgusemd関数は，*text_gr*で指定した画面の画面モードを取得/設定する。対象とする画面がグラフィック画面(0xC00000 ～ 0xDEFFFF)ならば0を，テキスト画面(0xE40000 ～ 0xE7FFFF)ならば1を*text_gr*に指定する。

*mode*には0～3あるいは−1を指定できる。それぞれの意味は次のとおり。

- 0　誰も使用していない
- 1　システムで使用(ソフトキー/電卓/RAM-DISK)
- 2　アプリケーションで使用中
- 3　アプリケーションで使用し，破壊されたまま
- −1　画面モードの取得

_iocs_tgusemd関数はIOCSコール0x0Eを発行することによって処理される。

戻り値――設定前の画面モードを返す。

規　格――*XC*

_iocs_timeasc

用　途――時刻を表す2進数形式のデータを文字列に変換する。

書　式――#include <sys/iocs.h>

```
int _iocs_timeasc (int time, char *addr);
```

解　説――_iocs_timeasc 関数は，*time* で指定した2進数形式の時刻データを "HH:MM:SS" の形式に変換し，*addr* で指定した領域へ格納する。

time に指定するビットフィールドの内容は次のとおり。

```
00000000 hhhhhhhh mmmmmmmm ssssssss
```

- ssssssss　秒 (00 ～ 59)
- mmmmmmmm　分 (00 ～ 59)
- hhhhhhhh　時 (00 ～ 23)

addr には文字列を格納するバッファの先頭アドレスを指定するが，このバッファは最低9バイト以上の領域を確保しなければならない。また，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

_iocs_timeasc 関数は IOCS コール 0x5B を発行することによって処理される。

戻り値――失敗した場合は−1を返す。

規　格――*XC*

T

_iocs_timebcd

用　　途——2進数の時刻を内部時計にセットできる形式に変換する。

書　　式——#include <sys/iocs.h>

```
int _iocs_timebcd (int time);
```

解　　説——_iocs_timebcd 関数は，*time* で指定した2進数形式の時刻を，_iocs_timeset 関数で使用できる形式に変換する。

time に指定するビットフィールドの内容は次のとおり。

```
00000000 hhhhhhhh mmmmmmmm ssssssss
```

- hhhhhhhh　　時 (00 ～ 23)
- mmmmmmmm　　分 (00 ～ 59)
- ssssssss　　秒 (00 ～ 59)

_iocs_timebcd 関数は IOCS コール 0x52 を発行することによって処理される。

戻 り 値——成功した場合は次のようなビットフィールドで返し，失敗した場合は負の値を返す。分/秒はいずれも BCD2 桁である。

```
0000TTTT HHHHHHHH MMMMMMMM SSSSSSSS
```

- SSSSSSSS　　秒 (00 ～ 59)
- MMMMMMMM　　分 (00 ～ 59)
- HHHHHHHH　　時 (00 ～ 23)
- TTTT　　1なら24時間計であることを示す

規　　格——*XC*

関連項目——_iocs_timeset

_iocs_timebin

用　　途——時刻をBCD形式から2進数形式へ変換する。

書　　式——#include <sys/iocs.h>

```
int _iocs_timebin (int time);
```

解　　説——_iocs_timebin関数は*time*の時刻データをBCD形式から2進数形式へ変換する。

*time*に指定するビットフィールドの内容は次のとおり。分/秒はいずれもBCD2桁である。

```
00000000 HHHHHHHH MMMMMMMM SSSSSSSS
```

- SSSSSSSS　　秒 (00 ~ 59)
- MMMMMMMM　　分 (00 ~ 59)
- HHHHHHHH　　時 (00 ~ 23)

_iocs_timebin関数はIOCSコール0x57を発行することによって処理される。

戻り値——次のようなビットフィールドを返す。分/秒はいずれも2進数である。

```
00000000 HHHHHHHH MMMMMMMM SSSSSSSS
```

- SSSSSSSS　　秒 (00 ~ 59)
- MMMMMMMM　　分 (00 ~ 59)
- HHHHHHHH　　時 (00 ~ 23)

規　　格——*XC*

T

_iocs_timecnv

用　途――時刻を表す文字列を2進数形式に変換する。

書　式――#include <sys/iocs.h>

```
int _iocs_timecnv (const char *addr);
```

解　説――_iocs_timecnv関数は，*addr*で指定した文字列を2進数形式の時刻データへ変換する。

*addr*には時刻を表す文字列の先頭アドレスを指定するが，このアドレスにはスーパーバイザ領域も指定できるので注意すること。

文字列は“HH:MM:SS”の形式で指定する。時刻の区切り記号は“:”でも，それ以外でもよい。

_iocs_timecnv関数はIOCSコール0x59を発行することによって処理される。

戻り値――成功した場合は次のようなビットフィールドを返し，失敗した場合は負の値を返す。

00000000 hhhhhhhh mmmmmmmm ssssssss

- ssssssss　　秒(00～59)
- mmmmmmmm　　分(00～59)
- hhhhhhhh　　時(00～23)

規　格――*XC*

_iocs_timeget

用　途——内部時計から時刻を読み込む。

書　式——#include <sys/iocs.h>

```
int _iocs_timeget (void);
```

解　説——_iocs_timeget 関数は内部時計から現在の時刻を読み込む。

_iocs_timeget 関数は IOCS コール 0x56 を発行することによって処理される。

戻り値——次のようなビットフィールドを返す。分/秒はすべて BCD2 桁である。

```
0000TTTT HHHHHHHH MMMMMMMM SSSSSSSS
```

- SSSSSSSS　秒 (00 ～ 59)
- MMMMMMMM　分 (00 ～ 59)
- HHHHHHHH　時 (00 ～ 23)
- TTTT　1 なら 24 時間計であることを示す

規　格——*XC*

T

_iocs_timerdst

用　　途——MFPのTIMER–Dによる割り込みを設定する。

書　　式——#include <sys/iocs.h>

```
int _iocs_timerdst (const void *addr, int mode, int cnt);
```

解　　説——_iocs_timerdst関数はMFPのTIMER-D割り込みを設定する。*addr*には割り込み処理アドレスを指定するが，0を指定することで割り込みを禁止できる。

*mode*には次の値を指定する。

- 1　　1 μ秒単位
- 2　　2.5 μ秒単位
- 3　　4 μ秒単位
- 4　　12.5 μ秒単位
- 5　　16 μ秒単位
- 6　　25 μ秒単位
- 7　　50 μ秒単位

*cnt*には，TIMER–D割り込みが何回ごとに設定した処理を行うかを，回数(0～255)で指定する。つまり，*mode*で指定される割り込み周期に*cnt*を乗じた単位で，割り込み処理が行われる。

_iocs_timerdst関数はIOCSコール0x6Bを発行することによって処理される。

戻り値——割り込みが設定された場合は0を返し，すでに使用している場合は0以外の値を返す。

注　　意——**Human68k**の動作中は割り込みを変更できない。

規　　格——*XC*

_iocs_timeset

用　途——内部時計に時刻を設定する。

書　式——#include <sys/iocs.h>

```
void _iocs_timeset (int time);
```

解　説——_iocs_timeset 関数は，*time* で指定した時刻を内部時計に設定する。*time* の内容は _iocs_timebcd 関数の戻り値を参照のこと。

_iocs_timeset 関数は IOCS コール 0x53 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

関連項目——_iocs_timebcd

T

_iocs_tpalet

用　途——テキストパレットを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_tpalet (int no, int code);
```

解　説——_iocs_tpalet 関数は，*no*(0 ～ 15) で指定したテキストパレットを *code*で指定したカラーに設定する。

*no*で指定するテキストパレット番号のうち，0 ～ 3 はテキストの表示色，4 ～ 7 はソフトキーボード/電卓の表示色その 1 (4 ～ 7 は同じコードをセット)，8 ～ 15 はソフトキーボード/電卓の表示色その 2 (8 ～ 15 は同じコードをセット) である。*no*が属する範囲内のテキストパレットは全て同じカラーに設定される。

*code*には 0 ～ 65536 までの値を指定するが，−1 を指定するとパレットの読み出しを行うことができ，また−2 を指定するとパレットをシステム規定値に戻すことができる。

_iocs_tpalet 関数は IOCS コール 0x13 を発行することによって処理される。

戻り値——*code*に−1 を指定したときだけ，*no*で指定したパレットの設定値を返す。

規　格——*XC*

_iocs_tpalet2

用　途──テキストパレットを設定する。

書　式──#include <sys/iocs.h>

```
int _iocs_tpalet2 (int no, int code);
```

解　説──_iocs_tpalet2 関数は，*no*(0 ～ 15) で指定したテキストパレットを *code* で指定したカラーに設定する。_iocs_tpalet2 関数は _iocs_tpalet 関数とは異なり，0 ～ 15 のテキストパレットそれぞれを自由に設定することができる。

code には 0 ～ 65536 までの値を指定するが，−1 を指定するとパレットの読み出しを行うことができ，また−2 を指定するとパレットをシステム規定値に戻すことができる。

_iocs_tpalet2 関数は IOCS コール 0x14 を発行することによって処理される。

戻り値──*code* に−1 を指定したときだけ，*no* で指定したパレットの設定値を返す。

規　格──*XC*

T

_iocs_trap15

用　途——内部割り込み (trap #15) を直接実行する。

書　式——#include <sys/iocs.h>

```
int _iocs_trap15 (const struct _regs *inregs,
                  struct _regs *outregs);
```

解　説——_iocs_trap15 関数は，*inregs* で指定した _regs 構造体のデータを各レジスタに複写し，直接 trap #15 を実行する。trap #15 実行後，各レジスタの内容を *outregs* で指定した _regs 構造体へ複写する。*inregs*，*outregs* に指定する _regs 構造体は次のとおり。

```
struct _regs {
    int d0; /* 実行時の各データレジスタに対応 */
    int d1;
    int d2;
    int d3;
    int d4;
    int d5;
    int d6;
    int d7;
    int a1; /* 実行時の各アドレスレジスタに対応 */
    int a2;
    int a3;
    int a4;
    int a5;
    int a6;
};
```

通常，trap #15 命令は IOCS コールを発行するために用いられるので，*inregs* のメンバ d0 に IOCS コール番号を設定し，それ以外のメンバに必要な値を設定してから _iocs_trap15 関数を実行すると，直接 IOCS コールを発行することができる。ただし，*inregs* に対して誤った値を設定すると暴走する可能性があるので十分に注意して使用すること。

戻り値——trap #15 命令実行後の d0 レジスタの値を返す。

規　格——*XC*

_iocs_tvctrl

用　途──専用テレビを操作する。

書　式──#include <sys/iocs.h>

```
void _iocs_tvctrl (int code);
```

解　説──_iocs_tvctrl 関数は，専用テレビに *code* で指定したコマンドを送信し，専用テレビのコントロールを行う。

code に指定するコマンドは次のとおり。

- 0x01　ボリュームを上げる
- 0x02　ボリュームを下げる
- 0x03　ボリュームを標準にする
- 0x04　チャンネルコール
- 0x05　テレビ画面 (初期化/リセット)
- 0x06　音声ミュート
- 0x07　電源 ON
- 0x08　テレビ，コンピュータ
- 0x09　テレビ，外部，コンピュータノーマル，スーパー
- 0x0A　コントラスト標準
- 0x0B　チャンネルアップ
- 0x0C　チャンネルダウン
- 0x0D　電源 OFF
- 0x0E　電源 ON/OFF
- 0x0F　スーパー 1
- 0x10　チャンネル 1
- 0x11　チャンネル 2
- 0x12　チャンネル 3
- 0x13　チャンネル 4
- 0x14　チャンネル 5
- 0x15　チャンネル 6
- 0x16　チャンネル 7
- 0x17　チャンネル 8
- 0x18　チャンネル 9
- 0x19　チャンネル 10
- 0x1A　チャンネル 11
- 0x1B　チャンネル 12
- 0x1C　テレビ画面 (0x05)
- 0x1D　コンピュータ画面 (0x05+0x08)

T

- 0x1E　　　　スーパー 1 (0x05+0x0F)
- 0x1F　　　　スーパー 2 (0x05+0x0F+0x0A)
- 0x21 ～ 0x3F　電源を入れた後，上記コマンドを実行 (0x20+0x01 ～ 0x1F)

_iocs_tvctrl 関数は IOCS コール 0x0C を発行することによって処理される。

戻り値——なし。

規　格——*XC*

_iocs_txbox

用　途――テキスト画面にボックスを描画する。

書　式――#include <sys/iocs.h>

```
void _iocs_txbox (const struct _tboxptr *ptr);
```

解　説――_iocs_txbox 関数はテキスト画面にボックスを描画する。*ptr* に指定する _tboxptr 構造体は次のとおり。

```
struct _tboxptr {
    unsigned short vram_page; /* テキストページ */
    short x;                  /* 始点の X 座標 */
    short y;                  /* 始点の Y 座標 */
    short x1;                 /* 終点の X 座標 */
    short y1;                 /* 終点の Y 座標 */
    unsigned short line_style; /* ラインスタイル */
};
```

_iocs_txbox 関数は IOCS コール 0xD6 を発行することによって処理される。

戻り値――なし。

規　格――*XC*

T

_iocs_txfill

用　　途——テキスト画面に塗りつぶしたボックスを描画する。

書　　式——#include <sys/iocs.h>

```
void _iocs_txfill (const struct _txfillptr *ptr);
```

解　　説——_iocs_txfill 関数はテキスト画面に塗りつぶしたボックスを描画する。*ptr*に指定する_txfillptr 構造体は次のとおり。

```
struct _txfillptr {
    unsigned short vram_page; /* テキストページ */
    short x;                  /* 始点の X 座標 */
    short y;                  /* 始点の Y 座標 */
    short x1;                 /* 終点の X 座標 */
    short y1;                 /* 終点の Y 座標 */
    unsigned short fill_patn; /* 塗りつぶしパターン */
};
```

_iocs_txfill 関数は IOCS コール 0xD7 を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_iocs_txrascpy

用　途——テキスト画面のラスタコピーを行う。

書　式——

```
#include <sys/iocs.h>
void _iocs_txrascpy (int sr_dst, int copy, int mode);
```

解　説——_iocs_txrascpy 関数はテキスト画面のラスタコピーを行う。

sr_dst にはソース× 256+ディスティネーションを指定する。各 8 ビットは 4 ラスタ単位で指定すること。また，*copy* には 4 ラスタを 1 単位として計算したコピーラスタ数を指定し，*mode* にはコピーする画面とそのコピー方向を指定する。画面は同時に 4 面まで指定可能である。

mode に指定できる値の内容は次のとおり。

- ビット 0　　テキスト 0
- ビット 1　　テキスト 1
- ビット 2　　テキスト 2
- ビット 3　　テキスト 3
- ビット 15　　コピー方向 (0 =インクリメント，1 =デクリメント)

_iocs_txrascpy 関数は IOCS コール 0xDF を発行することによって処理される。

戻り値——なし。

規　格——*XC*

T

_iocs_txrev

用　　途——テキスト画面を反転する。

書　　式——#include <sys/iocs.h>

```
void _iocs_txrev (const struct _trevptr *ptr);
```

解　　説——_iocs_txrev 関数はテキスト画面を反転する。*ptr* に指定する _trevptr 構造体は次のとおり。

```
struct _trevptr {
    unsigned short vram_page; /* テキストページ */
    short x;                  /* 始点の X 座標 */
    short y;                  /* 始点の Y 座標 */
    short x1;                 /* 終点の X 座標 */
    short y1;                 /* 終点の Y 座標 */
};
```

_iocs_txrev 関数は IOCS コール 0xD8 を発行することによって処理される。

戻 り 値——なし。

規　　格——*XC*

_iocs_txxline

用　途——テキスト画面に水平方向のラインを描画する。

書　式——#include <sys/iocs.h>

```
void _iocs_txxline (const struct _xlineptr *ptr);
```

解　説——_iocs_txxline 関数はテキスト画面に水平方向のラインを描画する。*ptr* に指定する _xlineptr 構造体は次のとおり。

```
struct _xlineptr {
    unsigned short vram_page; /* テキストページ */
    short x;                  /* 始点の X 座標 */
    short y;                  /* 始点の Y 座標 */
    short x1;                 /* 終点までの X の長さ */
    unsigned short line_style; /* ラインスタイル */
};
```

_iocs_txxline 関数は IOCS コール 0xD3 を発行することによって処理される。

戻り値——なし。

規　格——*XC*

T

_iocs_txyline

用　　途——テキスト画面に垂直方向のラインを描画する。

書　　式——#include <sys/iocs.h>

```
void _iocs_txyline (struct _ylineptr *ptr);
```

解　　説——_iocs_txyline 関数はテキスト画面に垂直方向のラインを描画する。*ptr* に指定する _ylineptr 構造体は次のとおり。

```
struct _ylineptr {
    unsigned short vram_page;  /* テキストページ */
    short x;                   /* 始点の X 座標 */
    short y;                   /* 始点の Y 座標 */
    short y1;                  /* 終点までの Y の長さ */
    unsigned short line_style; /* ラインスタイル */
};
```

_iocs_txyline 関数は IOCS コール 0xD4 を発行することによって処理される。

戻り値——なし。

規　　格——*XC*

_iocs_vdispst

用　途——垂直同期による割り込みを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_vdispst (const void *addr, int mode, int cnt);
```

解　説——_iocs_vdispst 関数は垂直同期割り込みを設定する。*addr*には割り込み処理アドレスを指定するが，0を指定することで割り込みを禁止できる。

*mode*には次の内容を指定する。

- 0　垂直帰線期間をカウント
- 1　垂直表示期間をカウント

*cnt*には，割り込みが何回ごとに設定した処理を行うかを，回数 (0 ~ 255) で指定する。つまり，*mode*で指定される割り込み周期に*cnt*を乗じた単位で，割り込み処理が行われる。

_iocs_vdispst 関数は IOCS コール 0x6C を発行することによって処理される。

戻り値——割り込みが設定された場合は0を返し，すでに使用している場合は0以外の値を返す。

規　格——*XC*

V

_iocs_vpage

用　途――グラフィック画面の表示ページを設定する。

書　式――

```
#include <sys/iocs.h>
int _iocs_vpage (int mode);
```

解　説――_iocs_vpage 関数はグラフィック画面の表示ページの設定を行う。

mode は表示ページを指定する。指定する値は次のとおり。それぞれ1で表示することを，0で表示しないことを表す。

- ビット0　0ページ
- ビット1　1ページ
- ビット2　2ページ
- ビット3　3ページ

_iocs_vpage 関数は IOCS コール 0xB2 を発行することによって処理される。

戻り値――成功した場合は0を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない
- −2　規定外のページを指定した
- −3　指定したページは現在の画面モードでは設定できない

規　格――*XC*

_iocs_window

用　途——グラフィック画面のウィンドウを設定する。

書　式——#include <sys/iocs.h>

```
int _iocs_window (int sx, int sy, int ex, int ey);
```

解　説——_iocs_window 関数はグラフィック画面のウィンドウの設定を行う。*sx*, *sy*, *ex*, *ey*にはそれぞれウィンドウに設定する左上，右下の座標を指定するが，指定する座標は必ず $sx \leq ex$, $sy \leq ey$の条件を満たさなければならない。いずれも 0 ～ 1023 の範囲で指定すること。

_iocs_window 関数は IOCS コール 0xB4 を発行することによって処理される。

戻り値——成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない
- −2　規定外の引数を指定した

規　格——*XC*

_iocs_wipe

用　途――グラフィック画面を消去する。

書　式――#include <sys/iocs.h>

```
int _iocs_wipe (void);
```

解　説――_iocs_wipe 関数はグラフィック画面を消去する。

_iocs_wipe 関数は IOCS コール 0xB5 を発行することによって処理される。

戻り値――成功した場合は 0 を返し，失敗した場合は次の値を返す。

- −1　グラフィックは使用できない

規　格――*XC*

Chapter 4
マルチバイト文字ライブラリ

本章にはマルチバイト文字ライブラリのマニュアルを掲載します。*LIBC*では*MS–C 7.0*と同等のマルチバイト文字関数群を提供しています。

ただし，これらのマルチバイト文字関数群はロケールに関してのサポートを一切しておらず，単にシフトJIS文字コードに対するサポートにとどめています。なぜならば*ANSI C*において，マルチバイト文字は規定されてはいるものの，実際の関数レベルではほとんどサポートされていないのが現状だからです。

*LIBC*ではマルチバイトへのアプローチとして，とりあえず*MS–C 7.0*互換の環境を提供します。よくマニュアルを読んで使用してください。

ismbbalnum

用　途——半角英数字/半角カタカナかどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbbalnum (int c);
#include <jctype.h>
int isalnmkana (int c);
```

解　説——ismbbalnum 関数は *c* で指定した文字コードが，半角英数字あるいは半角カタカナかどうかを調べる。結果は isalnum かつ ismbbkalnum。

戻り値——*c* が半角英数字か半角カタカナならば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbalnum 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbalnum 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbalnum 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——isalnmkana 関数は ismbbalnum 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，isalnmkana 関数を使用すること。

規　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——ismbbalpha, ismbbgraph, ismbbkalnum, ismbbkana, ismbbkpunct, ismbblead, ismbbprint, ismbbpunct, ismbbtrail

ismbbalpha

用　　途――半角英字/半角カタカナかどうかを調べる。

書　　式――

```
#include <mbctype.h>
int ismbbalpha (int c);
#include <jctype.h>
int isalkana (int c);
```

解　　説――ismbbalpha 関数は *c* で指定した文字コードが，半角英字あるいは半角カタカナかどうかを調べる。結果は isalpha かつ ismbbkalnum。

戻 り 値――*c* が半角英字か半角カタカナならば 0 以外の値を返し，異なれば 0 を返す。

注　　意――*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbalpha 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbalpha 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbalpha 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性――isalkana 関数は ismbbalpha 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，isalkana 関数を使用すること。

規　　格――***Project LIBC Group*，*MS–C 7.0***

関連項目――ismbbalnum, ismbbgraph, ismbbkalnum, ismbbkana, ismbbkpunct, ismbblead, ismbbprint, ismbbpunct, ismbbtrail

ismbbgraph

用　　途——半角スペース以外の表示可能文字かどうかを調べる。

書　　式——

```
#include <mbctype.h>
int ismbbgraph (int c);
#include <jctype.h>
int isgrkana (int c);
```

解　　説——ismbbgraph 関数は *c* で指定した文字コードが，半角スペース以外の表示可能文字（半角カナ文字を含む）かどうかを調べる。

戻 り 値——*c* が半角スペース以外の表示可能文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbgraph 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbgraph 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbgraph 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性——isgrkana 関数は ismbbgraph 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，isgrkana 関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——ismbbalnum，ismbbalpha，ismbbkalnum，ismbbkana，ismbbkpunct，ismbblead，ismbbprint，ismbbpunct，ismbbtrail

ismbbkalnum

用　途——半角カタカナかどうかを調べる。

書　式——
```
#include <mbctype.h>
int ismbbkalnum (int c);
#include <jctype.h>
int iskmoji (int c);
```

解　説——ismbbkalnum 関数は *c* で指定した文字コードが，半角カタカナ (半角カナ記号は含まない) かどうかを調べる。

戻り値——*c* が半角カタカナならば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbkalnum 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbkalnum 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbkalnum 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——iskmoji 関数は ismbbkalnum 関数に対する別名でる。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，iskmoji 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbbalnum, ismbbalpha, ismbbgraph, ismbbkana, ismbbkpunct, ismbblead, ismbbprint, ismbbpunct, ismbbtrail

ismbbkana

用　　途――半角カナ文字かどうかを調べる。

書　　式――

```
#include <mbctype.h>
int ismbbkana (int c);
#include <jctype.h>
int iskana (int c);
```

解　　説――ismbbkana関数は*c*で指定した文字コードが，半角カナ文字(半角カナ記号と半角カタカナ 0xA1 ～ 0xDF)かどうかを調べる。

戻 り 値――*c*が半角カナ文字ならば0以外の値を返し，異なれば0を返す。

注　　意――*c*はint型の引数だが，その値はunsigned char型で表現できる範囲の値かあるいはEOFと同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbkana関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbbkana関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbkana関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互 換 性――iskana関数はismbbkana関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，iskana関数を使用すること。

規　　格――***Project LIBC Group*，*MS-C 7.0***

関連項目――ismbbalnum, ismbbalpha, ismbbgraph, ismbbkalnum, ismbbkpunct, ismbblead, ismbbprint, ismbbpunct, ismbbtrail

ismbbkpunct

用　途――半角カナ記号かどうかを調べる。

書　式――
```
#include <mbctype.h>
int ismbbkpunct (int c);
#include <jctype.h>
int iskpun (int c);
```

解　説――ismbbkpunct 関数は *c* で指定した文字コードが，半角カナ記号かどうかを調べる。

戻り値――*c* が半角カナ記号ならば 0 以外の値を返し，異なれば 0 を返す。

注　意――*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbkpunct 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbkpunct 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbkpunct 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性――iskpun 関数は ismbbkpunct 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，iskpun 関数を使用すること。

規　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――ismbbalnum, ismbbalpha, ismbbgraph, ismbbkalnum, ismbbkana, ismbblead, ismbbprint, ismbbpunct, ismbbtrail

ismbblead

用　　途――全角文字の第1バイトかどうかを調べる。

書　　式――

```
#include <mbctype.h>
int ismbblead (int c);
#include <jctype.h>
int iskanji (int c);
```

解　　説――ismbblead 関数は *c* で指定した文字コードが，シフト JIS 全角文字の第1バイト (0x81 ～ 0x9F，0xE0 ～ 0xFC) かどうかを調べる。

戻 り 値――*c* がシフト JIS 全角文字の第1バイトならば0以外の値を返し，異なれば0を返す。

注　　意――*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbblead 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbblead 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbblead 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性――iskanji 関数は ismbblead 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，iskanji 関数を使用すること。

規　　格――***Project LIBC Group***，***MS–C 7.0***

関連項目――ismbbalnum，ismbbalpha，ismbbgraph，ismbbkalnum，ismbbkana，ismbbkpunct，ismbbprint，ismbbpunct，ismbbtrail

変　　更――従来では ismbblead 関数が真の値を返す値の範囲を 0x81 ～ 0x9F，0xE0 ～ 0xFC としていたが，現在の ***LIBC*** には 0x80 を追加した。**Human68k** では，0x80?? のコード部 FC 分に2バイト半角コードが割り当てられているためである。

ismbbprint

用　途——半角印字可能文字かどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbbprint (int c);
#include <jctype.h>
int isprkana (int c);
```

解　説——ismbbprint 関数は *c* で指定した文字コードが，半角印字可能文字かどうかを調べる。結果は isprint かつ ismbbkana。

戻り値——*c* が半角表示可能文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbprint 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbprint 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbprint 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——isprkana 関数は ismbbprint 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，isprkana 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbbalnum，ismbbalpha，ismbbgraph，ismbbkalnum，ismbbkana，ismbbkpunct，ismbblead，ismbbpunct，ismbbtrail

ismbbpunct

用　　途——半角記号/半角カナ記号かどうかを調べる。

書　　式——

```
#include <mbctype.h>
int ismbbpunct (int c);
#include <jctype.h>
int ispnkana (int c);
```

解　　説——ismbbpunct 関数は *c* で指定した文字コードが半角記号または半角カナ記号かどうかを調べる。結果は ispunct かつ ismbbkpunct。

戻 り 値——*c* が半角記号/半角カナ記号ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbpunct 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbpunct 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbpunct 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性——ispnkana 関数は ismbbpunct 関数に対する別名である。もしも ***XC*** や ***MS-C 6.0*** に対する互換性が必要ならば，ispnkana 関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——ismbbalnum，ismbbalpha，ismbbgraph，ismbbkalnum，ismbbkana，ismbbkpunct，ismbblead，ismbbprint，ismbbtrail

ismbbtrail

用　途——全角文字の第2バイトかどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbbtrail (int c);
#include <jctype.h>
int iskanji2 (int c);
```

解　説——ismbbtrail 関数は *c* で指定した文字コードが，シフト JIS 全角文字の第2バイト (0x40 ～ 0x7E，0x80 ～ 0xFC) かどうかを調べる。

戻り値——*c* がシフト JIS 全角文字の第2バイトならば0以外の値を返し，異なれば0を返す。

注　意——*c* は int 型の引数だが，その値は unsigned char 型で表現できる範囲の値かあるいは EOF と同値でなければならない。この範囲を越える部分については，その動作は未定義である。

通常，ismbbtrail 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbbtrail 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbbtrail 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——iskanji2 関数は ismbbtrail 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，iskanji2 関数を使用すること。

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——ismbbalnum，ismbbalpha，ismbbgraph，ismbbkalnum，ismbbkana，ismbbkpunct，ismbblead，ismbbprint，ismbbpunct

ismbcalpha

用　　途——全角英字かどうかを調べる。

書　　式——

```
#include <mbctype.h>
int ismbcalpha (int c);
#include <jctype.h>
int jisalpha (int c);
```

解　　説——ismbcalpha関数はcで指定した文字コードが，全角英字 (0x8260 ～ 0x8279, 0x8281 ～ 0x829A) かどうかを調べる。cは全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻 り 値——cが全角英字ならば0以外の値を返し，異なれば0を返す。

注　　意——通常，ismbcalpha関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbcalpha関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcalpha関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互 換 性——jisalpha関数はismbcalpha関数に対する別名である。もしも*XC*や*MS–C 6.0*に対する互換性が必要ならば，jisalpha関数を使用すること。

規　　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——ismbcdigit, ismbchira, ismbckata, ismbcl0, ismbcl1, ismbcl2, ismbclegal, ismbclower, ismbcprint, ismbcspace, ismbcsymbol, ismbcupper

ismbcdigit

用　　途——全角数字かどうかを調べる。

書　　式——#include <mbctype.h>

```
int ismbcdigit (int c);
#include <jctype.h>
int jisdigit (int c);
```

解　　説——ismbcdigit 関数は *c* で指定した文字コードが，全角数字 (0x824F ～ 0x8258) かどうかを調べる。*c* は全角文字の第 1 バイトを上位 8 ビットに，第 2 バイトを下位 8 ビットに設定すること。

戻 り 値——*c* が全角数字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——通常，ismbcdigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbcdigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcdigit 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性——jisdigit 関数は ismbcdigit 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jisdigit 関数を使用すること。

規　　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——ismbcalpha，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

I

ismbchira

用　　途――全角ひらがなかどうかを調べる。

書　　式――
```
#include <mbctype.h>
int ismbchira (int c);
#include <jctype.h>
int jishira (int c);
```

解　　説――ismbchira関数は*c*で指定した文字コードが，全角ひらがな(0x829F ～ 0x82F1)かどうかを調べる。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻 り 値――*c*が全角ひらがなならば0以外の値を返し，異なれば0を返す。

注　　意――通常，ismbchira関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbchira関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbchira関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互 換 性――jishira関数はismbchira関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，jishira関数を使用すること。

規　　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――ismbcalpha，ismbcdigit，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbckata

用　途——全角カタカナかどうかを調べる。

書　式——
```
#include <mbctype.h>
int ismbckata (int c);
#include <jctype.h>
int jiskata (int c);
```

解　説——ismbckata 関数は *c* で指定した文字コードが，全角カタカナ (0x8340 ～ 0x8396) かどうかを調べる。*c* は全角文字の第 1 バイトを上位 8 ビットに，第 2 バイトを下位 8 ビットに設定すること。

戻り値——*c* が全角カタカナならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，ismbckata 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbckata 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbckata 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——jiskata 関数は ismbckata 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，jiskata 関数を使用すること。

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbcl0

用　　途——JIS 非漢字かどうかを調べる。

書　　式——

```
#include <mbctype.h>
int ismbcl0 (int c);
#include <jctype.h>
int jisl0 (int c);
```

解　　説——ismbcl0 関数は *c* で指定した文字コードが，JIS 非漢字 (0x8140 ～ 0x889E) かどうかを調べる。*c* は全角文字の第 1 バイトを上位 8 ビットに，第 2 バイトを下位 8 ビットに設定すること。

戻 り 値——*c* が JIS 非漢字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——通常，ismbcl0 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbcl0 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcl0 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互 換 性——jisl0 関数は ismbcl0 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jisl0 関数を使用する。

規　　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbcalpha, ismbcdigit, ismbchira, ismbckata, ismbcl1, ismbcl2, ismbclegal, ismbclower, ismbcprint, ismbcspace, ismbcsymbol, ismbcupper

ismbcl1

用　途——JIS 第 1 水準漢字かどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbcl1 (int c);
#include <jctype.h>
int jisl1 (int c);
```

解　説——ismbcl1 関数は *c* で指定した文字コードが，JIS 第 1 水準漢字 (0x889F ～ 0x9872) かどうかを調べる。*c* は全角文字の第 1 バイトを上位 8 ビットに，第 2 バイトを下位 8 ビットに設定すること。

戻り値——*c* が JIS 第 1 水準漢字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，ismbcl1 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbcl1 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcl1 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——jisl1 関数は ismbcl1 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jisl1 関数を使用すること。

規　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbcl2

用　途——JIS 第2水準漢字かどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbcl2 (int c);
#include <jctype.h>
int jisl2 (int c);
```

解　説——ismbcl2 関数は *c* で指定した文字コードが，JIS 第2水準漢字 (0x989F ～ 0xEA9E) かどうかを調べる。*c* は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値——*c* が JIS 第2水準漢字ならば0以外の値を返し，異なれば0を返す。

注　意——通常，ismbcl2 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbcl2 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcl2 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——jisl2 関数は ismbcl2 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，jisl2 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbclegal

用　途——正しいシフトJIS全角文字かどうかを調べる。

書　式——
```
#include <mbctype.h>
int ismbclegal (int c);
#include <jctype.h>
int jiszen (int c);
```

解　説——ismbclegal 関数は *c* で指定した文字コードが，シフトJISコードとして正しいかどうかを調べる。*c* は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値——*c* が正しいシフトJIS全角文字ならば0以外の値を返し，異なれば0を返す。

注　意——通常，ismbclegal 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbclegal 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbclegal 関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性——jiszen 関数は ismbclegal 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，jiszen 関数を使用すること。

規　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclower，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbclower

用　途——全角英小文字かどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbclower (int c);
#include <jctype.h>
int jislower (int c);
```

解　説——ismbclower関数は*c*で指定した文字コードが，全角英小文字(0x8281～0x829A)かどうかを調べる。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値——*c*が全角英小文字ならば0以外の値を返し，異なれば0を返す。

注　意——通常，ismbclower関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbclower関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbclower関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性——jislower関数はismbclower関数に対する別名である。もしも*XC*や***MS–C 6.0***に対する互換性が必要ならば，jislower関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbcprint，ismbcspace，ismbcsymbol，ismbcupper

ismbcprint

用　途──印字可能文字かどうかを調べる。

書　式──
```
#include <mbctype.h>
int ismbcprint (int c);
#include <jctype.h>
int jisprint (int c);
```

解　説──ismbcprint 関数は c で指定した文字コードが，印字可能文字 (半角カナ文字，全角文字を含む) かどうかを調べる。結果は isprint かつ iskana かつ ismbclegal。c は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値──c が印字可能文字ならば0以外の値を返し，異なれば0を返す。

注　意──通常，ismbcprint 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbcprint 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcprint 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性──jisprint 関数は ismbcprint 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jisprint 関数を使用すること。

規　格──*Project LIBC Group*，*MS–C 7.0*

関連項目──ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcspace，ismbcsymbol，ismbcupper

ismbcspace

用　途——全角スペースかどうかを調べる。

書　式——

```
#include <mbctype.h>
int ismbcspace (int c);
#include <jctype.h>
int jisspace (int c);
```

解　説——ismbcspace関数は*c*で指定した文字コードが，全角スペース(0x8140)かどうかを調べる。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値——*c*が全角スペースならば0以外の値を返し，異なれば0を返す。

注　意——通常，ismbcspace関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbcspace関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcspace関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性——jisspace関数はismbcspace関数に対する別名である。もしも*XC*や*MS–C 6.0*に対する互換性が必要ならば，jisspace関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcsymbol，ismbcupper

ismbcsymbol

用　途——全角記号かどうかを調べる。

書　式——#include <mbctype.h>

```
int ismbcsymbol (int c);
#include <jctype.h>
int jiskigou (int c);
```

解　説——ismbcsymbol 関数は *c* で指定した文字コードが，全角記号 (0x8141 ～ 0x81AC) かどうかを調べる。*c* は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値——*c* が全角記号ならば0以外の値を返し，異なれば0を返す。

注　意——通常，ismbcsymbol 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。ismbcsymbol 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcsymbol 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——jiskigou 関数は ismbcsymbol 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jiskigou 関数を使用すること。

規　格——*Project LIBC Group*，*MS–C 7.0*

関連項目——ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcupper

ismbcupper

用　途――全角英大文字かどうかを調べる。

書　式――

```
#include <mbctype.h>
int ismbcupper (int c);
#include <jctype.h>
int jisupper (int c);
```

解　説――ismbcupper関数は*c*で指定した文字コードが，全角英大文字(0x8260～0x8279)かどうかを調べる。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値――*c*が全角英大文字ならば0以外の値を返し，異なれば0を返す。

注　意――通常，ismbcupper関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。ismbcupper関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

ismbcupper関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性――jisupper関数はismbcupper関数に対する別名である。もしも*XC*や*MS–C 6.0*に対する互換性が必要ならば，jisupper関数を使用すること。

規　格――*Project LIBC Group*，*MS–C 7.0*

関連項目――ismbcalpha，ismbcdigit，ismbchira，ismbckata，ismbcl0，ismbcl1，ismbcl2，ismbclegal，ismbclower，ismbcprint，ismbcspace，ismbcsymbol

mbbtombc

用　途——半角文字を全角文字に変換する。

書　式——

```
#include <mbstring.h>
int mbbtombc (int c);
#include <jstring.h>
int hantozen (int c);
```

解　説——mbbtombc 関数は *c* で指定した半角文字を，対応する全角文字に変換する。変換対象となるコードは 0x20 ～ 0x7E，0xA1 ～ 0xDF であり，それ以外の文字は変化しない。

戻り値——*c* が変換可能ならばその変換結果を返し，不可ならば *c* をそのまま返す。

注　意——mbbtombc 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性——hantozen 関数は mbbtombc 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，hantozen 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——mbctombb

M

mbbtype

用　途――バイトデータの文字種を判別する。

書　式――
```
#include <mbstring.h>
int mbbtype (unsigned char c, int type);
#include <jstring.h>
int chkctype (unsigned char c, int type);
```

解　説――mbbtype関数は*c*で指定したバイトデータの文字種を判別し，その結果を返す。*type*が1以外の値を指定した場合は，*c*が半角文字かシフトJIS全角文字の第1バイトかどうかを判別し，*type*が1ならば*c*がシフトJIS全角文字の第2バイトかどうかを判別する。

戻り値――*type*が1以外の場合，*c*の文字種を返す。結果は次のマクロ値をとる。

- MBC_SINGLE　　半角文字(0x20 ～ 0x7E，0xA1 ～ 0xFC)
- MBC_LEAD　　シフトJIS全角文字の第1バイト
- MBC_ILLEGAL　　それ以外(null文字を含む)

*type*が1の場合，*c*の文字種を返す。結果は次のマクロ値をとる。

- MBC_TRAIL　　シフトJIS全角文字の第2バイト
- MBC_ILLEGAL　　それ以外(null文字を含む)

注　意――mbbtype関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性――chkctype関数はmbbtype関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，chkctype関数を使用すること。

規　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbsbtype

mbctohira

用　途――全角カタカナを全角ひらがなに変換する。

書　式――

```
#include <mbstring.h>
int mbctohira (int c);
#include <jstring.h>
int jtohira (int c);
```

解　説――mbctohira 関数は *c* で指定された全角カタカナの文字コードを，全角ひらがなの文字コードに変換する。*c* が全角カタカナでなかったり，変換できない場合は *c* をそのまま返す。*c* は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値――*c* が変換可能ならばその変換結果を返し，不可ならば *c* をそのまま返す。

注　意――通常，mbctohira 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。mbctohira 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

mbctohira 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性――jtohira 関数は mbctohira 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，jtohira 関数を使用すること。

規　格――***Project LIBC Group***，***MS–C 7.0***

関連項目――mbbtombc，mbctokata，mbctolower，mbctombb，mbctoupper

M

mbctokata

用　途――全角ひらがなを全角カタカナに変換する。

書　式――
```
#include <mbstring.h>
int mbctokata (int c);
#include <jstring.h>
int jtokata (int c);
```

解　説――mbctokata関数は*c*で指定した全角ひらがなの文字コードを，全角カタカナの文字コードに変換する。*c*が全角ひらがなでなかったり，変換できない場合は*c*をそのまま返す。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻り値――*c*が変換可能ならばその変換結果を返し，不可ならば*c*をそのまま返す。

注　意――通常，mbctokata関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。mbctokata関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

mbctokata関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互換性――jtokata関数はmbctokata関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jtokata関数を使用すること。

規　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbbtombc，mbctohira，mbctolower，mbctombb，mbctoupper

mbctolower

用　途――全角英大文字を全角英小文字に変換する。

書　式――

```
#include <mbstring.h>
int mbctolower (int c);
#include <jstring.h>
int jtolower (int c);
```

解　説――mbctolower 関数は *c* で指定した全角英大文字の文字コードを，全角英小文字の文字コードに変換する。*c* が全角英大文字でなかったり，変換できない場合は *c* をそのまま返す。*c* は全角文字の第１バイトを上位８ビットに，第２バイトを下位８ビットに設定すること。

戻り値――*c* が変換可能ならばその変換結果を返し，不可ならば *c* をそのまま返す。

注　意――通常，mbctolower 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。mbctolower 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

mbctolower 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性――jtolower 関数は mbctolower 関数に対する別名である。もしも *XC* や *MS–C 6.0* に対する互換性が必要ならば，jtolower 関数を使用すること。

規　格――*Project LIBC Group*，*MS–C 7.0*

関連項目――mbbtombc，mbctohira，mbctokata，mbctombb，mbctoupper

M

mbctombb

用　　途——全角文字を半角文字に変換する。

書　　式——
```
#include <mbstring.h>
int mbctombb (int c);
#include <jstring.h>
int zentohan (int c);
```

解　　説——mbctombb関数は*c*で指定したシフトJIS全角文字を，対応する半角文字に変換する。変換対象となるコードは，結果が0x20～0x7E，0xA1～0xDFとなるようなコードのみで，それ以外の文字は変化しない。

戻 り 値——*c*が変換可能ならばその変換結果を返し，不可ならば*c*をそのまま返す。

注　　意——mbctombb関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。なお濁点や半濁点のついた文字は，これらを除いた文字に変換される。

互 換 性——zentohan関数はmbctombb関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，zentohan関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbbtombc

mbctoupper

用　　途——全角英小文字を全角英大文字に変換する。

書　　式——
```
#include <mbstring.h>
int mbctoupper (int c);
#include <jstring.h>
int jtoupper (int c);
```

解　　説——mbctoupper関数は*c*で指定した全角英小文字の文字コードを，全角英大文字の文字コードに変換する。*c*が全角英小文字でなかったり，変換できない場合は*c*をそのまま返す。*c*は全角文字の第1バイトを上位8ビットに，第2バイトを下位8ビットに設定すること。

戻 り 値——*c*が変換可能ならばその変換結果を返し，不可ならば*c*をそのまま返す。

注　　意——通常，mbctoupper関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。mbctoupper関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

mbctoupper関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。

互 換 性——jtoupper関数はmbctoupper関数に対する別名である。もしも***XC***や***MS–C 6.0***に対する互換性が必要ならば，jtoupper関数を使用すること。

規　　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——mbbtombc，mbctohira，mbctokata，mbctolower，mbctombb

mbsbtype

用　途――文字列中の指定位置の文字種を判別する。

書　式――
```
#include <mbstring.h>
int mbsbtype (const unsigned char *s, size_t n);
#include <jstring.h>
int nthctype (const unsigned char *s, size_t n);
```

解　説――mbsbtype 関数は *s* で指定した文字列の *n* バイト目の文字種を判別し，その結果を返す。ただし，*n* バイト目までに null 文字があった場合は MBC_ILLEGAL を返す。

戻り値――*s* で指定した文字列の *n* バイト目の文字種を返す。結果は次のマクロ値をとる。

- MBC_SINGLE　半角文字 (0x20 ～ 0x7E，0xA1 ～ 0xFC)
- MBC_LEAD　シフト JIS 全角文字の第1バイト
- MBC_TRAIL　シフト JIS 全角文字の第2バイト
- MBC_ILLEGAL　それ以外 (null 文字を含む)

注　意――mbsbtype 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。

互換性――nthctype 関数は mbsbtype 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，nthctype 関数を使用すること。

規　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――mbbtype

mbscat

用　途——シフト JIS 文字列を他のシフト JIS 文字列に連結する。

書　式——

```
#include <mbstring.h>
unsigned char *mbscat (unsigned char *mbstring1,
                       const unsigned char *mbstring2);
#include <jstring.h>
unsigned char *jstrcat (unsigned char *mbstring1,
                        const unsigned char *mbstring2);
```

解　説——`mbscat` 関数は *mbstring2* の指す文字列を *mbstring1* の指す文字列の最後にコピーし，連結した文字列の最後に null 文字をつける。*mbstring1* の末尾の null 文字には，*mbstring2* の最初の文字を重ねてコピーする。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値——*mbstring1* へのポインタを返す。

M

注　意——`mbscat` 関数はシフト JIS コードを扱うためのものであり，**ロケール**とは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

mbstring1 は，連結した結果の文字列を格納するのに十分な領域を指していなければならない。

互換性——`jstrcat` 関数は `mbscat` 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば `jstrcat` 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——`mbsncat`

mbschr

用　途――シフトJIS文字列中から指定文字を検索する。

書　式――
```
#include <mbstring.h>
unsigned char *mbschr (const unsigned char *mbstring,
                       int character);
#include <jstring.h>
unsigned char *jstrchr (const unsigned char *mbstring,
                        int character);
```

解　説――mbschr関数は*mbstring*が指す文字列中から，*character*で指定した文字を検索する。*mbstring*末尾のnull文字も検索の対象となり得る。*character*に2バイト文字を指定するときは，1バイト目を上位8ビットに，2バイト目を下位8ビットに設定すること。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――*mbstring*中で最初に現れた*character*へのポインタを返す。見つからない場合はNULLを返す。

注　意――mbschr関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性――jstrchr関数はmbschr関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jstrchr関数を使用すること。

規　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbscspn，mbspbrk，mbsrchr，mbsspn，mbsstr，mbstok

mbscmp

用　　途――2つのシフトJIS文字列を比較する。

書　　式――#include <mbstring.h>

```
int mbscmp (const unsigned char *mbstring1,
            const unsigned char *mbstring2);
#include <jstring.h>
int jstrcmp (const unsigned char *mbstring1,
             const unsigned char *mbstring2);
```

解　　説――mbscmp関数は，*mbstring1*の指す文字列と*mbstring2*の指す文字列とを比較する。比較はnull文字が検出された時点で終了する。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻 り 値――比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での*mbstring1*側の文字が*mbstring2*側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意――mbscmp関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互 換 性――jstrcmp関数はmbscmp関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，jstrcmp関数を使用すること。

規　　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbsncmp

M

mbscpy

用　　途――シフト JIS 文字列をコピーする。

書　　式――
```
#include <mbstring.h>
unsigned char *mbscpy (unsigned char *mbstring1,
                       const unsigned char *mbstring2);
#include <jstring.h>
unsigned char *jstrcpy (unsigned char *mbstring1,
                        const unsigned char *mbstring2);
```

解　　説――mbscpy 関数は *mbstring2* の指す文字列を，null 文字を含めて *mbstring1* の指す領域にコピーする。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値――*mbstring1* へのポインタを返す。

注　　意――mbscpy 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

領域が重なっていた場合，その動作は未定義である。また *mbstring1* は，*mbstring2* の指す文字列を格納するのに十分な領域を指していなければならない。

互 換 性――jstrcpy 関数は mbscpy 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，jstrcpy 関数を使用すること。

規　　格――***Project LIBC Group，MS–C 7.0***

関連項目――mbsncpy

mbscspn

用　　途──指定したシフト JIS 文字列に含まれない文字が，ほかのシフト JIS 文字列の先頭から何文字続いているかを調べる。

書　　式──
```
#include <mbstring.h>
size_t mbscspn (const unsigned char *mbstring1,
                const unsigned char *mbstring2);
#include <jstring.h>
size_t jstrcspn (const unsigned char *mbstring1,
                 const unsigned char *mbstring2);
```

解　　説──mbscspn 関数は *mbstring1* の指す文字列と *mbstring2* の指す文字列とを比較し，*mbstring1* の先頭から *mbstring2* に含まれない文字で構成されている部分の長さを調べる。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値──*mbstring1* の先頭から，*mbstring2* に含まれない文字で構成されている部分の長さを返す。

注　　意──mbscspn 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性──jstrcspn 関数は mbscspn 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，jstrcspn 関数を使用すること。

規　　格──***Project LIBC Group*，*MS-C 7.0***

関連項目──mbschr，mbspbrk，mbsrchr，mbsspn，mbsstr，mbstok

M

mbsdec

用　　途――シフトJIS文字列のポインタを1文字分戻す。

書　　式――#include <mbstring.h>

```
unsigned char *mbsdec (const unsigned char *mbstring,
                       unsigned char *ptr);
```

解　　説――mbsdec関数は*mbstring*が指す文字列のなかで，*ptr*が指す位置から1文字分戻った位置を求めて，その位置へのポインタを返す。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――*mbstring*が指定する文字列中で，*ptr*が指定する位置から1文字戻った位置へのポインタを返す。もし文字列の先頭で，これ以上戻れない場合はNULLを返す。

注　　意――mbsdec関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

規　　格――***Project LIBC Group，MS-C 7.0***

関連項目――mbsinc，mbsnextc

mbsdup

用　　途――新しい領域を確保してシフトJIS文字列をコピーする。

書　　式――
```
#include <mbstring.h>
unsigned char *mbsdup (const unsigned char *mbstring);
#include <jstring.h>
unsigned char *jstrdup (const unsigned char *mbstring);
```

解　　説――mbsdup関数は*mbstring*の指す文字列を，null文字を含めてmalloc関数によって確保した領域にコピーする。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――正常にコピーできた場合はコピー先へのポインタを返し，失敗した場合はNULLを返して，変数errnoにその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　　意――mbsdup関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性――jstrdup関数はmbsdup関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，jstrdup関数を使用すること。

規　　格――***Project LIBC Group*，*MS-C 7.0***

関連項目――mbscpy

M

mbsicmp

用　　途——2つのシフトJIS文字列を大文字/小文字を区別せずに比較する。

書　　式——
```
#include <mbstring.h>
int mbsicmp (const unsigned char *mbstring1,
             const unsigned char *mbstring2);
#include<jstring.h>
int jstricmp (const unsigned char *mbstring1,
              const unsigned char *mbstring2);
```

解　　説——mbsicmp関数は*mbstring1*の指す文字列と*mbstring2*の指す文字列とを，大文字と小文字を区別しないで比較する。処理はnull文字が検出された時点で終了する。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻 り 値——比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での*mbstring1*側の文字が*mbstring2*側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意——mbsicmp関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

mbsicmp関数は1バイト文字を大文字にしてから比較するが，2バイト文字に関しては大文字変換は行わない。

互 換 性——jstricmp関数はmbsicmp関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jstricmp関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbscmp，mbsncmp

mbsinc

用　　途——シフト JIS 文字列のポインタを 1 文字分進める。

書　　式——

```
#include <mbstring.h>
unsigned char *mbsinc (const unsigned char *mbstring,
                                  unsigned char *ptr);
```

解　　説——mbsinc 関数は *mbstring* が指す文字列のなかで，*ptr* が指す位置から 1 文字分進んだ位置を求めて，その位置へのポインタを返す。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値——*mbstring* が指定する文字列中で，*ptr* が指定する位置から 1 文字進んだ位置へのポインタを返す。もし文字列の終端で，これ以上進めない場合は NULL を返す。

注　　意——mbsinc 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

規　　格——***Project LIBC Group，MS-C 7.0***

関連項目——mbsdec，mbsnextc，mbsninc

mbslen

用　途――シフトJIS文字列の長さを調べる。

書　式――
```
#include <mbstring.h>
size_t mbslen (const unsigned char *mbstring);
#include <jstring.h>
size_t jstrlen (const unsigned char *mbstring);
```

解　説――mbslen関数は，*mbstring*の指す文字列の末尾のnull文字を除いた文字列の長さを調べる。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――末尾のnull文字を除いた文字列の長さを返す。

注　意――mbslen関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性――jstrlen関数はmbslen関数に対する別名である。もしも*XC*や*MS-C6.0*に対する互換性が必要ならば，jstrlen関数を使用すること。

規　格――*Project LIBC Group*，*MS-C 7.0*

関連項目――mbscpy

mbslwr

用　　途——シフト JIS 文字列を小文字に変換する。

書　　式——#include <mbstring.h>

```
unsigned char *mbslwr (unsigned char *mbstring);
#include <jstring.h>
unsigned char *jstrlwr (unsigned char *mbstring);
```

解　　説——mbslwr 関数は *mbstring* が指す文字列を先頭から調べ，現在のロケールの LC_CTYPE カテゴリを基に大文字を小文字に変換する。処理は null 文字が検出された時点で終了する。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値——*mbstring* へのポインタを返す。

注　　意——mbslwr 関数はシフト JIS コードを扱うためのものであり，2バイト文字に関してはロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

mbslwr 関数が小文字に変換するのはシフト JIS 文字列中の1バイト文字だけで，2バイト文字に関しては変換されない。

互 換 性——jstrlwr 関数は mbslwr 関数に対する別名である。もしも ***XC*** や ***MS-C 6.0*** に対する互換性を必要とするならば，jstrlwr 関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbsnset，mbsrev，mbsset，mbsupr

M

mbsnbcnt

用　　途――シフトJIS文字列のバイト数を調べる。

書　　式――
```
#include <mbstring.h>
size_t mbsnbcnt (const unsigned char *mbstring, size_t n);
#include <jstring.h>
size_t mtob (const unsigned char *mbstring, size_t n);
```

解　　説――mbsnbcnt関数は*mbstring*が指す文字列の先頭から，*n*文字が実際に何バイトあるのかを求め，そのバイト数を返す。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――*mbstring*が指定する文字列の*n*文字目までのバイト数を返すが，*n*文字目に到達する前にnull文字があった場合はそこまでのバイト数となる。

注　　意――mbsnbcnt関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性――mtob関数はmbsnbcnt関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，mtob関数を使用すること。

規　　格――***Project LIBC Group，MS-C 7.0***

関連項目――mbsnccnt

mbsncat

用　途——シフト JIS 文字列を指定文字数だけほかのシフト JIS 文字列に追加する。

書　式——

```
#include <mbstring.h>
unsigned char *mbsncat (unsigned char *mbstring1,
                        const char *mbstring2, size_t n);
#include <jstring.h>
unsigned char *jstrncat (unsigned char *mbstring1,
                         const char *mbstring2, size_t n);
```

解　説——mbsncat 関数は *mbstring2* の指す文字列を *mbstring1* の指す文字列の最後にコピーし，連結した文字列の最後に null 文字をつける。*mbstring1* の末尾の null 文字には，*mbstring2* の最初の文字を重ねてコピーする。処理は null 文字が検出されるか，*n* 文字コピーした時点で終了する。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値——*mbstring1* へのポインタを返す。

注　意——mbsncat 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

mbstring1 は，連結した結果の文字列を格納するのに十分な領域を指していなければならない。

互換性——jstrncat 関数は mbsncat 関数に対する別名である。もしも ***XC*** や ***MS-C 6.0*** に対する互換性が必要ならば，jstrncat 関数を使用すること。

規　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbscat

M

mbsnccnt

用　　途──シフトJIS文字列の文字数を調べる。

書　　式──
```
#include <mbstring.h>
size_t mbsnccnt (const unsigned char *mbstring, size_t n);
#include <jstring.h>
size_t btom (const unsigned char *mbstring, size_t n);
```

解　　説──mbsnccnt関数は*mbstring*が指す文字列の先頭から，*n*バイトが実際に何文字になるか求め，その文字数を返す。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値──*mbstring*が指定する文字列の*n*バイト目までの文字数を返すが，*n*バイト目に到達する前にnull文字があった場合は，そこまでの文字数となる。

注　　意──mbsnccnt関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性──btom関数はmbsnccnt関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，btom関数を使用すること。

規　　格──*Project LIBC Group*，*MS-C 7.0*

関連項目──mbsnbcnt

mbsncmp

用　　途——2つのシフトJIS文字列を指定文字数だけ比較する。

書　　式——

```
#include <mbstring.h>
int mbsncmp (const unsigned char *mbstring1,
                const unsigned char *mbstring2, size_t n);
#include <jstring.h>
int jstrncmp (const unsigned char *mbstring1,
                const unsigned char *mbstring2, size_t n);
```

解　　説——mbsncmp関数は，*mbstring1*の指す文字列と*mbstring2*の指す文字列とを比較する。処理はnull文字が検出されるか，*n*文字比較した時点で終了する。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値——比較の結果，2つの文字列がまったく同じならば0を返す。異なっていたり*n*文字を比較し終えた場合，その位置での*mbstring1*側の文字が*mbstring2*側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意——mbsncmp関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性——jstrncmp関数はmbsncmp関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，mbsncmp関数を使用すること。

規　　格——*Project LIBC Group*，*MS-C 7.0*

関連項目——mbscmp

mbsncpy

用　　途──シフトJIS文字列を指定文字数だけコピーする。

書　　式──

```
#include <mbstring.h>
unsigned char *mbsncpy (unsigned char *mbstring1,
                        const unsigned char *mbstring2,
                        size_t n);
#include <jstring.h>
unsigned char *jstrncpy (unsigned char *mbstring1,
                         const unsigned char *mbstring2,
                         size_t n);
```

解　　説──mbsncpy関数は*mbstring2*の指す文字列の最初の*n*文字までを，*mbstring1*の指す領域にコピーする。処理はnull文字が検出されるか，*n*文字コピーした時点で終了する。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻 り 値──*mbstring1*へのポインタを返す。

注　　意──mbsncpy関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

領域が重なっていた場合，その動作は未定義である。また，*mbstring1*は*n*文字を格納するのに十分な領域を指していなければならない。

互 換 性──jstrncpy関数はmbsncpy関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jstrncpy関数を使用すること。

規　　格──***Project LIBC Group***，***MS-C 7.0***

関連項目──mbscpy

mbsnextc

用　途――ポインタが指す位置文字を返す。

書　式――#include <mbstring.h>

```
int mbsnextc (const unsigned char *ptr);
```

解　説――mbsnextc 関数は *ptr* が指す位置から1マルチバイト文字を取り出し，その値を返す。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値――*ptr* が指定する位置の1マルチバイト文字を返すが，この文字が2バイト文字だった場合は1バイト目を上位8ビットに，2バイト目を下位8ビットに格納する。

注　意――mbsnextc 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

規　格――***Project LIBC Group，MS–C 7.0***

関連項目――mbsdec，mbsinc，mbsninc

M

mbsninc

用　　途——シフトJIS文字列のポインタを指定文字分だけ進める。

書　　式——

```
#include <mbstring.h>
unsigned char *mbsninc (const unsigned char *mbstring,
                        size_t n);
#include <jstring.h>
unsigned char *jstradv (const unsigned char *mbstring,
                        size_t n);
```

解　　説——mbsninc関数は*mbstring*が指す文字列の*n*文字目の位置を求めて，その位置へのポインタを返す。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻 り 値——*mbstring*が指定する文字列の*n*文字目の位置を求め，その位置へのポインタを返す。もし文字列の終端で，これ以上進めない場合はNULLを返す。

注　　意——mbsninc関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互 換 性——jstradv関数はmbsninc関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jstradv関数を使用すること。

規　　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbsinc，mbsnextc

mbsnset

用　　途――シフト JIS 文字列を指定文字で指定文字数だけ埋める。

書　　式――#include <mbstring.h>

```
unsigned char *mbsnset (unsigned char *mbstring,
                        int character, size_t n);
#include <jstring.h>
unsigned char *jstrnset (unsigned char *mbstring,
                         int character, size_t n);
```

解　　説――mbsnset 関数は *mbstring* が指す文字列の先頭から，*n* 文字か null 文字を検出するまで，*character* で指定した文字で埋める。*character* に 2 バイト文字を指定するときは，1 バイト目を上位 8 ビットに，2 バイト目を下位 8 ビットに設定すること。

もしも *character* に 2 バイト文字を設定した場合，文字列領域が奇数バイトならば最後の 1 バイトは 1 バイトスペースで埋められる。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値――*mbstring* へのポインタを返す。

注　　意――mbsnset 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

n は文字列領域の大きさを越えてはならない。

互 換 性――jstrnset 関数は mbsnset 関数に対する別名である。もしも ***XC*** や ***MS-C 6.0*** に対する互換性が必要ならば，jstrnset 関数を使用すること。

規　　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbslen，mbslwr，mbsrev，mbsset，mbsupr

M

mbspbrk

用　　途――指定したシフトJIS文字列に含まれる文字が，ほかのシフトJIS文字列に存在するかどうかを調べる。

書　　式――

```
#include <mbstring.h>
unsigned char *mbspbrk (const unsigned char *mbstring1,
                        const unsigned char *mbstring2);
#include <jstring.h>
unsigned char *jstrpbrk (const unsigned char *mbstring1,
                         const unsigned char *mbstring2);
```

解　　説――mbspbrk 関数は *mbstring1* の指す文字列と *mbstring2* の指す文字列とを比較し，*mbstring2* に含まれる文字が最初に現れる場所を検索する。

文字列の最後は null 文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値――*mbstring1* 中で，*mbstring2* に含まれる文字が最初に現れる位置へのポインタを返す。見つからない場合は NULL を返す。

注　　意――mbspbrk 関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互 換 性――jstrpbrk 関数は mbspbrk 関数に対する別名である。もしも *XC* や *MS-C 6.0* に対する互換性が必要ならば，jstrpbrk 関数を使用すること。

規　　格――***Project LIBC Group*，*MS-C 7.0***

関連項目――mbschr，mbscspn，mbsrchr，mbsspn，mbsstr，mbstok

mbsrchr

用　　途――シフト JIS 文字列中から指定文字が最後に現れる位置を検索する。

書　　式――

```
#include <mbstring.h>
unsigned char *mbsrchr (const unsigned char *mbstring,
                        int character);
#include <jstring.h>
unsigned char *jstrrchr (const unsigned char *mbstring,
                         int character);
```

解　　説――mbsrchr 関数は *mbstring* が指す文字列中から，*character* で指定された文字が最後に現れる位置を検索する。*mbstring* 末尾の null 文字も検索の対象となり得る。*character* に 2 バイト文字を指定するときは，1 バイト目を上位 8 ビットに，2 バイト目を下位 8 ビットに設定すること。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻 り 値――*mbstring* 中で *character* が最後に現れる位置へのポインタを返す。見つからない場合は NULL を返す。

注　　意――mbsrchr 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

互 換 性――jstrrchr 関数は mbsrchr 関数に対する別名である。もしも ***XC*** や ***MS–C 6.0*** に対する互換性が必要ならば，jstrrchr 関数を使用すること。

規　　格――***Project LIBC Group***，***MS–C 7.0***

関連項目――mbschr，mbscspn，mbspbrk，mbsspn，mbsstr，mbstok

M

mbsrev

用　　途——シフトJIS文字列を前後反転させる。

書　　式——

```
#include <mbstring.h>
unsigned char *mbsrev (unsigned char *mbstring);
#include <jstring.h>
unsigned char *jstrrev (unsigned char *mbstring);
```

解　　説——mbsrev関数は*mbstring*が指す文字列を前後反転させる。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻 り 値——*mbstring*へのポインタを返す。

注　　意——mbsrev関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互 換 性——jstrrev関数はmbsrev関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，jstrrev関数を使用すること。

規　　格——***Project LIBC Group，MS-C 7.0***

関連項目——mbslwr，mbsnset，mbsset，mbsupr

mbsset

用　途——シフトJIS文字列を指定文字で埋める。

書　式——
```
#include <mbstring.h>
unsigned char *mbsset (unsigned char *mbstring,
                       int character);
#include <jstring.h>
unsigned char *jstrset (unsigned char *mbstring,
                        int character);
```

解　説——mbsset関数は*mbstring*が指す文字列の先頭から，null文字に出会うまで*character*で指定した文字で埋める。*character*に2バイト文字を指定するときは，1バイト目を上位8ビットに，2バイト目を下位8ビットに設定すること。

もしも*character*に2バイト文字を設定した場合，文字列領域が奇数バイトならば，最後の1バイトは1バイトスペースで埋められる。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値——*mbstring*へのポインタを返す。

注　意——mbsset関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性——jstrset関数はmbsset関数に対する別名である。もしも*XC*や*MS-C 6.0*に対する互換性が必要ならば，jstrset関数を使用すること。

規　格——***Project LIBC Group***，***MS-C 7.0***

関連項目——mbslwr，mbsnset，mbsrev，mbsupr

M

mbsspn

用　途——指定したシフト JIS 文字列に含まれる文字が，ほかのシフト JIS 文字列の先頭から何文字続いているかを調べる。

書　式——

```
#include <mbstring.h>
size_t mbsspn (const unsigned char *mbstring1,
               const unsigned char *mbstring2);
#include <jstring.h>
size_t jstrspn (const unsigned char *mbstring1,
                const unsigned char *mbstring2);
```

解　説——mbsspn 関数は *mbstring1* の指す文字列と *mbstring2* の指す文字列とを比較し，*mbstring1* の先頭から *mbstring2* に含まれる文字で構成されている部分の長さを調べる。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値——*mbstring1* の先頭から，*mbstring2* に含まれる文字で構成されている部分の長さを返す。

注　意——mbsspn 関数はシフト JIS コードを扱うためのものであり，ロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

互換性——jstrspn 関数は mbsspn 関数に対する別名である。もしも *XC* や ***MS–C 6.0*** に対する互換性が必要ならば，jstrspn 関数を使用すること。

規　格——***Project LIBC Group***，***MS–C 7.0***

関連項目——mbschr，mbscspn，mbspbrk，mbsrchr，mbsstr，mbstok

mbsstr

用　途——指定したシフトJIS文字列がほかのシフトJIS文字列に存在するかどうかを調べる。

書　式——
```
#include <mbstring.h>
size_t mbsstr (const unsigned char *mbstring1,
               const unsigned char *mbstring2);
#include <jstring.h>
size_t jstrstr (const unsigned char *mbstring1,
                const unsigned char *mbstring2);
```

解　説——mbsstr関数は*mbstring1*の指す文字列と*mbstring2*の指す文字列とを比較し，*mbstring1*のなかで，文字列*mbstring2*が最初に現れる位置を検索する。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値——*mbstring1*中で，*mbstring2*が最初に現れる位置へのポインタを返す。

注　意——mbsstr関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

互換性——jstrstr関数はmbsstr関数に対する別名である。もしも***XC***や***MS-C 6.0***に対する互換性が必要ならば，jstrstr関数を使用すること。

規　格——***Project LIBC Group，MS-C 7.0***

関連項目——mbschr，mbscspn，mbspbrk，mbsrchr，mbsspn，mbstok

M

mbstok

用　途――シフトJIS文字列を指定した区切り文字でトークンに分ける。

書　式――
```
#include <mbstring.h>
unsigned char *mbstok (unsigned char *mbstring,
                       const unsigned char *delim);
#include <jstring.h>
unsigned char *jstrtok (unsigned char *mbstring,
                        const unsigned char *delim);
```

解　説――mbstok関数を連続して呼ぶことによって，*mbstring*が指す文字列を語句(トークン)に分解することができる。各語句は，*delim*が指す文字列に含むまれる区切り文字(デリミタ)によって区切られる。

最初にmbstok関数を呼び出すときは，デリミタ文字列を*mbstring*に指定し，以降mbstok関数を呼び出すときには，*mbstring*にNULLを指定する。デリミタ文字列は，mbstok関数を呼び出すたびに変更することができる。*mbstring*にデリミタ文字列が続く場合は無視される。

文字列の最後はnull文字で判別されるが，不正なシフトJISマルチバイト文字があった場合はそれもnull文字とみなす。

戻り値――語句が見つかれば最後の区切り記号の位置にnull文字を設定し，語句の最初の文字へのポインタを返す。見つからなければNULLを返す。

注　意――mbstok関数はシフトJISコードを扱うためのものであり，ロケールとは関係がない。また，1文字は1マルチバイト文字を意味する。

mbstok関数は与えられた*mbstring*の領域を書き換えるので，元のデータが必要な場合は自分で保存する必要がある。

互換性――jstrtok関数はmbstok関数に対する別名である。もしも*XC*や***MS-C 6.0***に対する互換性が必要ならば，jstrtok関数を使用すること。

規　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbschr，mbscspn，mbspbrk，mbsrchr，mbsspn，mbsstr

mbsupr

用　途――シフト JIS 文字列を大文字に変換する。

書　式――
```
#include <mbstring.h>
unsigned char *mbsupr (unsigned char *mbstring);
#include <jstring.h>
unsigned char *jstrupr (unsigned char *mbstring);
```

解　説――mbsupr 関数は *mbstring* が指す文字列を先頭から調べ，現在の**ロケール**の LC_CTYPE カテゴリを基に小文字を大文字に変換する。処理は null 文字を検出した時点で終了する。

文字列の最後は null 文字で判別されるが，不正なシフト JIS マルチバイト文字があった場合はそれも null 文字とみなす。

戻り値――*mbstring* へのポインタを返す。

注　意――mbsupr 関数はシフト JIS コードを扱うためのものであり，2 バイト文字に関してはロケールとは関係がない。また，1 文字は 1 マルチバイト文字を意味する。

mbsupr 関数が大文字に変換するのはシフト JIS 文字列中の 1 バイト文字だけで，2 バイト文字に関しては変換されない。

互換性――jstrupr 関数は mbsupr 関数に対する別名である。もしも ***XC*** や ***MS-C 6.0*** に対する互換性が必要ならば，jstrupr 関数を使用すること。

規　格――***Project LIBC Group***，***MS-C 7.0***

関連項目――mbslwr，mbsnset，mbsrev，mbsset

M

Chapter 5

SCSI コールライブラリ

本章には Human68k の SCSI コールライブラリのマニュアルを掲載します。*LIBC* では従来の *XC* との互換性を考慮し，名前が異なる以外，基本的な動作はすべて同じになるように作成してあります。

本書では関数ごとのマニュアルにとどめ，SCSI コールおよび SCSI 自体についての詳しい説明は特にしていません。これらについてより詳しく知りたい場合は，他の書籍などを参照してください。

_scsi_cmdout

用　途——コマンドアウトフェーズを実行する。

書　式——#include <sys/scsi.h>

```
#include <sys/scsi.h>
int _scsi_cmdout (int n, const void *cdb);
```

解　説——_scsi_cmdout 関数は，*cdb*が指す領域から*n*バイトをコマンドとしてSCSIバス上に出力し，コマンドアウトフェーズを実行する。

戻り値——正常終了した場合は0を返し，失敗した場合はその原因を示すSCSIエラーコードを返す。

注　意——_scsi_cmdout 関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——***Project LIBC Group***，***XC***

関連項目——_scsi_select，_scsi_testunit

_scsi_datain

用　途——データインフェーズを実行する。

書　式——

```
#include <sys/scsi.h>
int _scsi_datain (int n, void *buff);
```

解　説——_scsi_datain 関数はデータインフェーズを実行して，SCSI バス上から DMA 転送を用いて *n* バイトのデータを読み込み，*buff* が指す領域に格納する。

戻り値——正常終了した場合は 0 を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　意——_scsi_datain 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

buff は，最低でも *n* 以上の大きさの領域を指していなければならない。

規　格——*Project LIBC Group*，*XC*

関連項目——_scsi_datain_p，_scsi_dataout，_scsi_dataout_p，_scsi_select，_scsi_testunit

_scsi_datain_p

用　途――データインフェーズを実行する。

書　式――#include <sys/scsi.h>

```
int _scsi_datain_p (int n, void *buff);
```

解　説――_scsi_datain_p 関数はデータインフェーズを実行して，SCSIバス上からプログラム転送を用いて *n* バイトのデータを読み込み，*buff* が指す領域に格納する。

戻り値――正常終了した場合は0を返し，失敗した場合はその原因を示すSCSIエラーコードを返す。

注　意――_scsi_datain_p 関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

buff は，最低でも *n* 以上の大きさの領域を指していなければならない。

規　格――*Project LIBC Group*

関連項目――_scsi_datain，_scsi_dataout，_scsi_dataout_p，_scsi_select，_scsi_testunit

_scsi_dataout

用　　途——データアウトフェーズを実行する。

書　　式——#include <sys/scsi.h>

```
int _scsi_dataout (int n, const void *buff);
```

解　　説——_scsi_dataout 関数はデータアウトフェーズを実行して，DMA 転送を用いて，*buff* が指す領域から *n* バイトを SCSI バス上に書き込む。

戻 り 値——正常終了した場合は 0 を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　　意——_scsi_dataout 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格——*Project LIBC Group*，*XC*

関連項目——_scsi_datain,　_scsi_datain_p,　_scsi_dataout_p,　_scsi_select,　_scsi_testunit

_scsi_dataout_p

用　途——データアウトフェーズを実行する。

書　式——#include <sys/scsi.h>

```
int _scsi_dataout_p (int n, const void *buff);
```

解　説——_scsi_dataout_p 関数はデータアウトフェーズを実行して，プログラム転送を用いて，*buff* が指す領域から *n* バイトを SCSI バス上に書き込む。

戻り値——正常終了した場合は 0 を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　意——_scsi_dataout_p 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——*Project LIBC Group*

関連項目——_scsi_datain, _scsi_datain_p, _scsi_dataout, _scsi_select, _scsi_testunit

_scsi_format

用　　途——SCSIユニットを物理フォーマットする。

書　　式——#include <sys/scsi.h>

```
int _scsi_format (int shift, int id);
```

解　　説——_scsi_format関数は*id*で指定したSCSI IDのSCSIユニットを，*shift*で指定したインタリーブ値で物理フォーマットする。

戻 り 値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意——_scsi_format関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格——***Project LIBC Group，XC***

関連項目——_scsi_select，_scsi_testunit

_scsi_inquiry

用　　途――INQUIRYデータを要求する。

書　　式――#include <sys/scsi.h>

```
int _scsi_inquiry (int n, int id, struct _inquiry *buff);
```

解　　説――_scsi_inquiry関数は*id*で指定したSCSI-IDユニットから，INQUIRYデータを*n*バイト読み込んで，*buff*が指す領域に格納する。

戻 り 値――正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意――_scsi_inquiry関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも*n*以上の大きさの領域を指していなければならない。

規　　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select，_scsi_testunit

_scsi_modeselect

用　途——SCSI ユニットにモードセレクトコマンドを発行する。

書　式——#include <sys/scsi.h>

```
int _scsi_modeselect (int mode, int len,
                      int id, const void *buff);
```

解　説——_scsi_modeselect 関数は，*buff*が指す領域から *len*バイトをパラメータとして SCSI-ID ユニットに書き込む。また *mode*には，PF(ページフォーマット)，SP(セーブパラメータ) のビットを指定する。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_modeselect 関数はSCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——*Project LIBC Group*，*XC*

関連項目——_scsi_select，_scsi_testunit

_scsi_modesense

用　　途——SCSIユニットの各種パラメータを調べる。

書　　式——#include <sys/scsi.h>

```
int _scsi_modesense (int page,
                     int n, int id, void *buff);
```

解　　説——_scsi_modesense関数は*page*(PCF/ページコード)を指定して，*id*で指定したSCSI IDのSCSIユニットのパラメータを*n*(アロケーションレングス)バイト読み込み，*buff*が指す領域に格納する。

戻 り 値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意——_scsi_modesense関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも*n*以上の大きさの領域を指していなければならない。

規　　格——***Project LIBC Group***，***XC***

関連項目——_scsi_select，_scsi_testunit

_scsi_msgin

用　途——メッセージインフェーズを実行する。

書　式——#include <sys/scsi.h>

```
int _scsi_msgin (void *buff);
```

解　説——_scsi_msgin 関数はメッセージインフェーズを実行して，*buff*が指す領域に1バイトのデータを SCSI バス上から読み込む。

戻り値——正常終了した場合は0を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　意——_scsi_msgin 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも1バイト以上の大きさの領域を指していなければならない。

規　格——*Project LIBC Group*，*XC*

関連項目——_scsi_select，_scsi_testunit

M

_scsi_msgout

用　　途——メッセージアウトフェーズを実行する。

書　　式——#include <sys/scsi.h>

```
int _scsi_msgout (const void *buff);
```

解　　説——_scsi_msgout 関数はメッセージアウトフェーズを実行して，*buff*が指す領域から1バイトのデータを SCSI バス上に書き込む。

戻 り 値——正常終了した場合は0を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　　意——_scsi_msgout 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格——*Project LIBC Group*，*XC*

関連項目——_scsi_select，_scsi_testunit

_scsi_pamedium

用　　途――SCSI ユニットにメディアのイジェクトの禁止/許可を設定する。

書　　式――#include <sys/scsi.h>

```
int _scsi_pamedium (int mode, int id);
```

解　　説――_scsi_pamedium 関数は *id* で指定した SCSI ID の SCSI ユニットに，イジェクトのモードを設定する。*mode* が 0 ならばイジェクト許可，1 ならばイジェクト禁止となる。

戻り値――正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意――_scsi_pamedium 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select，_scsi_testunit

P

_scsi_phase

用　途——SCSIフェーズセンスを実行する。

書　式——#include <sys/scsi.h>

```
int _scsi_phase (void);
```

解　説——_scsi_phase関数は，現在実行しているフェーズを**SPC**(SCSIプロトコルコントローラ)のPSNSレジスタから読み出す。

戻り値——PSNSレジスタの内容を返す。

注　意——_scsi_phase関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——*Project LIBC Group*, *XC*

関連項目——_scsi_select, _scsi_testunit

_scsi_read

用　　途——SCSI ユニットのデータを読み込む。

書　　式——
```
#include &lt;sys/scsi.h&gt;
int _scsi_read (int pos, int blk,
                int id, int size, void *buff);
```

解　　説——_scsi_read 関数は *id* で指定した SCSI ID の SCSI ユニットの *pos*(論理ブロック番号) の位置から，*blk*(論理ブロック数) ×*size*(0=256，1=512，2=1024) バイトを読み込み，*buff* が指す領域に格納する。

戻 り 値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意——_scsi_read 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

buff は，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　　格——*Project LIBC Group*，*XC*

関連項目——_scsi_select，_scsi_testunit

R

_scsi_readcap

用　途——SCSIユニットの容量に関する情報を読み込む。

書　式——#include <sys/scsi.h>

```
int _scsi_readcap (int id, _readcap *buff);
```

解　説——_scsi_readcap関数は*id*で指定したSCSI IDのSCSIユニットのREAD CAPACITYデータを8バイト分，*buff*が指す領域に読み込む。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_readcap関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも8バイト以上の大きさの領域を指していなければならない。

規　格——*Project LIBC Group*，*XC*

関連項目——_scsi_select，_scsi_testunit

_scsi_readext

用　途——拡張 READ コマンドで SCSI ユニットのデータを読み込む。

書　式——#include <sys/scsi.h>

```
int _scsi_readext (int pos, int blk,
                        int id, int size, void *buff);
```

解　説——_scsi_readext 関数は *id* で指定した SCSI ID の SCSI ユニットの *pos*(論理ブロック番号) の位置から，*blk*(論理ブロック数) ×*size*(0=256，1=512，2=1024) バイトを読み込み，*buff* が指す領域に格納する。論理ブロック数は 65535 ブロックまで指定可能。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_readext 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

buff は，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　格——***Project LIBC Group，XC***

関連項目——_scsi_select，_scsi_testunit

R

_scsi_reassign

用　途——SCSIユニットの欠陥ブロックの再割り当てを行う。

書　式——#include <sys/scsi.h>

```
int _scsi_reassign (int n, int id, const void *buff);
```

解　説——_scsi_reassign関数は*buff*が指す領域から*n*バイトを，REASSIGN BLOCKSコマンドにより，*id*で指定したSCSI IDのSCSIユニットに書き込む。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_reassign関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——***Project LIBC Group***，***XC***

関連項目——_scsi_select，_scsi_testunit

_scsi_request

用　途——SCSIユニットのセンスデータを調べる。

書　式——#include <sys/scsi.h>

```
int _scsi_request (int n, int id, void *buff);
```

解　説——_scsi_request関数は，*id*で指定したSCSI IDのSCSIユニットのREQUEST SENSEデータを*n*バイト分読み込み，*buff*が指す領域に格納する。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_request関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも*n*以上の大きさの領域を指していなければならない。

規　格——***Project LIBC Group***，***XC***

関連項目——_scsi_select，_scsi_testunit

R

_scsi_reset

用　途――**SPC**(SCSIプロトコルコントローラ)のリセットおよびSCSIバスのリセットを行う。

書　式――#include <sys/scsi.h>

```
void _scsi_reset (void);
```

解　説――_scsi_reset関数は**SPC**(SCSIプロトコルコントローラ)のリセットおよびSCSIバスのリセットを実行し，バスリセット後2秒間待機する。

戻り値――なし。

注　意――_scsi_reset関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select，_scsi_testunit

_scsi_rezerounit

用　途——SCSI ユニットを指定の状態にセットする。

書　式——#include <sys/scsi.h>

```
int _scsi_rezerounit (int id);
```

解　説——_scsi_rezerounit 関数は，*id* で指定した SCSI ID の SCSI ユニットを指定した状態にセットする。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_rezerounit 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　格——***Project LIBC Group*，*XC***

関連項目——_scsi_select，_scsi_testunit

R

_scsi_seek

用　　途――SCSIユニットを指定の論理ブロックにシークする。

書　　式――#include <sys/scsi.h>

```
int _scsi_seek (int pos, int id);
```

解　　説――_scsi_seek関数は，*id*で指定したSCSI IDのSCSIユニットを*pos*(論理ブロック番号)の位置にシークする。

戻り値――正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意――_scsi_seek関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――***Project LIBC Group***，***XC***

関連項目――_scsi_select，_scsi_testunit

_scsi_select

用　　途――アービトレーションフェーズとセレクションフェーズを実行する。

書　　式――
```
#include <sys/scsi.h>
int _scsi_select (int id);
```

解　　説――_scsi_select 関数は *id* で指定した SCSI ID の SCSI ユニットに対して，アービトレーションフェーズとセレクションフェーズを実行する。

戻 り 値――正常終了した場合は 0 を返し，失敗した場合はその原因を示す SCSI エラーコードを返す。

注　　意――_scsi_select 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――***Project LIBC Group*, *XC***

関連項目――_scsi_testunit

_scsi_startstop

用　　途――SCSIユニットに対して，以降の操作を可能または不可能にする。

書　　式――#include <sys/scsi.h>

```
int _scsi_startstop (int mode, int id);
```

解　　説――_scsi_startstop関数は，*id*で指定したSCSI IDのSCSIユニットの操作モードを*mode*とする。*mode*が0ならば操作不可能，1ならば操作可能となる。

戻り値――正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意――_scsi_startstop関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――***Project LIBC Group***，***XC***

関連項目――_scsi_select，_scsi_testunit

_scsi_stsin

用　途――ステータスインフェーズを実行する。

書　式――#include <sys/scsi.h>

```
int _scsi_stsin (void *buff);
```

解　説――_scsi_stsin 関数はステータスインフェーズを実行し，*buff*が指す領域に1バイトのデータをSCSIバス上から読み込む。

戻り値――正常終了した場合は0を返し，失敗した場合はその原因を示すSCSIエラーコードを返す。

注　意――_scsi_stsin 関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

*buff*は，最低でも1バイト以上の大きさの領域を指していなければならない。

規　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select，_scsi_testunit

S

_scsi_testunit

用　　途――SCSIユニットが動作可能かどうかを調べる。

書　　式――#include <sys/scsi.h>

```
int _scsi_testunit (int id);
```

解　　説――_scsi_testunit関数は，*id*で指定したSCSI IDのSCSIユニットが動作可能かどうかを調べる。

戻り値――動作可能ならば0を返し，それ以外ならばステータス情報を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　　意――_scsi_testunit関数はSCSIデバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select

_scsi_write

用　途——SCSI ユニットにデータを書き込む。

書　式——

```
#include <sys/scsi.h>
int _scsi_write (int pos, int blk,
                 int id, int size, const void *buff);
```

解　説——_scsi_write 関数は *buff* が指す領域のデータを，*id* で指定した SCSI ID の SCSI ユニットの *pos*(論理ブロック番号) の位置から，*blk*(論理ブロック数)×*size*(0=256, 1=512, 2=1024) バイト分書き込む。

戻り値——正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

注　意——_scsi_write 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

buff は，結果を格納するだけの十分な大きさの領域を指していなければならない。

規　格——***Project LIBC Group***，***XC***

関連項目——_scsi_select，_scsi_testunit

W

_scsi_writeext

用　　途――拡張 WRITE コマンドで SCSI ユニットにデータを書き込む。

書　　式――
```
#include <sys/scsi.h>
int _scsi_writeext (int pos, int blk,
                    int id, int size, const void *buff);
```

解　　説――_scsi_writeext 関数は *buff* が指す領域のデータを，*id* で指定した SCSI ID の SCSI ユニットの *pos* (論理ブロック番号) の位置から，*blk* (論理ブロック数) × *size* (0=256, 1=512, 2=1024) バイト分書き込む。論理ブロック数は 65535 ブロックまで指定可能。

戻り値――正常終了した場合はステータス情報を返し，失敗した場合は負の値を返す。ステータス情報は上位がメッセージインフェーズで得た情報，下位がステータスインフェーズで得た情報である。

buff は，結果を格納するだけの十分な大きさの領域を指していなければならない。

注　　意――_scsi_writeext 関数は SCSI デバイスドライバを登録することにより，初めて使用することができる。登録されていない場合，その動作は一切保証されないので注意すること。

規　　格――*Project LIBC Group*，*XC*

関連項目――_scsi_select，_scsi_testunit

Chapter 6
幅広文字ライブラリ

本章には幅広文字 (ワイドキャラクタ) ライブラリのマニュアルを掲載します。*LIBC*では*MSE*(Multibyte Support Extension)や*SVR4*, *OSF/1*で規定されている幅広文字関数群のうち，有用と思われる約40の関数を提供しています。

*ANSI C*でも幅広文字についてはまだ取り入れられただけであり，詳しい関数の規格などは現在検討されている状況です。しかし現実レベルでは*SVR4*, *OSF/1*，そして*MSE*などでどんどん幅広文字関数が取り入れられています。*LIBC*ではこういったことをふまえ，完全ではないにしろ幅広文字をサポートすることにしました。

基本的には普通の文字列を扱う関数と同じですから，慣れてしまえば初心者にとっても扱いやすいものです。よくマニュアルを読んで使用してください。ただし，これらの幅広文字に関しては将来正式な規格が制定された段階で，変更されることがあります。あらかじめ注意してください。

fgetwc

用　途——ファイルストリームから１幅広文字を取り出す。

書　式——

```
#include <widec.h>
wint_t fgetwc (FILE *stream);
```

解　説——fgetwc関数は，*stream*で指定されたファイルストリームから１幅広文字をwchar_t型のデータとして取り出し，それをwint_t型に変換して返す。ストリームのファイルポインタは，これに応じて１幅広文字分進む。

戻り値——正常に取り出せた場合はそのデータを返す。もしストリーム操作中に，何らかのエラーが発生した場合はストリームの**エラー指示子**を設定してWEOFを返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定してWEOFを返す。失敗した場合は，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　格——*Project LIBC Group*，*SYSV*

関連項目——getwc

fgetws

用　途——ファイルストリームから幅広文字列を取り出す。

書　式——

```
#include <widec.h>
char *fgetws (wchar_t *s, int n, FILE *stream);
```

解　説——fgetws 関数は *stream* で指定されたファイルストリームから幅広文字列を取り出し，結果を *s* が指す領域に格納する。幅広文字列の取り出しは $n-1$ 幅広文字を処理するか，ファイルの終端に到達するか，**改行文字**を処理するまで続けられる。結果の幅広文字列には改行文字が含まれ，最後に null 文字が置かれることに注意すること。

戻り値——正常に幅広文字列を取り出せた場合は *s* を返す。もしストリーム操作中に，何らかのエラーが発生した場合はストリームの**エラー指示子**を設定して NULL を返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定して NULL を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——*s* は少なくとも *n* 幅広文字分の領域を指していなければならない。

規　格——***Project LIBC Group，SYSV***

関連項目——ferror，fread，getws

fputwc

用　途——ファイルストリームに幅広文字を出力する。

書　式——#include <widec.h>

```
#include <widec.h>
wint_t fputwc (wint_t wc, FILE *stream);
```

解　説——fputwc 関数は *stream* で指定されたファイルストリームに対して，幅広文字 *wc* を wchar_t 型に変換して出力する。これに応じて，ストリームのファイルポインタは1幅広文字分進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

戻り値——正常に出力できた場合は *wc* を返し，何らかのエラーによって失敗した場合は，ストリームのエラー指示子を設定して WEOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクが一杯である

規　格——*Project LIBC Group*，*SYSV*

関連項目——ferror，fopen，putwc，putws

fputws

用　途——ファイルストリームに幅広文字列を出力する。

書　式——

```
#include <widec.h>
int fputws (const wchar_t *s, FILE *stream);
```

解　説——fputws 関数は *stream* で指定されたファイルストリームに対して，*s* が指す幅広文字列を出力する。幅広文字列は null 文字までとし，最後の null 文字は出力しない。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

戻り値——正常に出力できた場合は負ではない値を返し，何らかのエラーによって失敗した場合はストリームのエラー指示子を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　格——*Project LIBC Group*，*SYSV*

関連項目——ferror，fopen，putwc，putws

getwc

用　　途——ファイルストリームから1幅広文字を取り出す。

書　　式——#include <widec.h>

```
wint_t getwc(FILE *stream);
```

解　　説——getwc関数は，*stream*で指定されたファイルストリームから1幅広文字をwchar_t型のデータとして取り出し，それをwint_t型に変換して返す。ストリームのファイルポインタは，これに応じて1幅広文字分進む。

戻 り 値——正常に取り出せた場合はそのデータを返す。もしストリーム操作中に，何らかのエラーが発生した場合はストリームのエラー指示子を設定してWEOFを返し，ファイルの終端に到達した場合はストリームの終端指示子を設定してWEOFを返す。失敗した場合は，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　　意——getwc関数はつねにfgetwc関数の別名である。

規　　格——***Project LIBC Group*，*SYSV***

関連項目——fgetwc

getwchar

用　途——標準入力ファイルストリームから１幅広文字を取り出す。

書　式——#include <widec.h>

```
wint_t getwchar (void);
```

解　説——getwchar 関数は，標準入力に割り当てられたファイルストリームから１幅広文字を wchar_t 型のデータとして取り出し，それを wint_t 型に変換して返す。ストリームのファイルポインタは，これに応じて１幅広文字分進む。

getwchar 関数は，fgetwc 関数に stdin を指定した場合と同じことである。

戻り値——正常に取り出せた場合はそのデータを返す。もしストリーム操作中に，何らかのエラーが発生した場合は標準入力ストリームのエラー指示子を設定して WEOF を返し，ファイルの終端に到達した場合はストリームの終端指示子を設定して WEOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——getwchar 関数はつねにマクロとして定義される。

規　格——*Project LIBC Group*，*SYSV*

関連項目——fgetwc，getwc

getws

用　途——標準入力ファイルストリームから幅広文字列を取り出す。

書　式——#include <widec.h>

```
wchar_t *getws (wchar_t *s);
```

解　説——getws関数は標準入力に割り当てられたファイルストリームから幅広文字列を取り出し，結果をsが指す領域に格納する。幅広文字列の取り出しはファイルの終端に到達するか，**改行文字**に到達するまで続けられる。結果の幅広文字列には改行文字は含まれず，最後にnull文字が置かれることに注意すること。

戻り値——正常に幅広文字列を取り出せた場合にはsを返す。もしストリーム操作中に，何らかのエラーが発生した場合はストリームの**エラー指示子**を設定してNULLを返し，ファイルの終端に到達した場合はストリームの**終端指示子**を設定してNULLを返す。失敗した場合は，変数errnoにその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

注　意——getws関数はfgetws関数とは異なり，結果の幅広文字列に改行文字を含まないことに注意する。またsは結果を格納するのに十分な領域を指していなければならない。

規　格——*Project LIBC Group*，*SYSV*

関連項目——ferror，fread

iswalnum

用　途——英数字かどうかを調べる。

書　式——`#include <wwctype.h>`

```
int iswalnum (wint_t c);
```

解　説——iswalnum 関数は *c* で指定した文字コードを，現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**英数字**であるかどうかを調べる。

戻り値——*c* が英数字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，iswalnum 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswalnum 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*SYSV*

関連項目——iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswupper，iswxdigit

I

iswalpha

用　　途——アルファベットかどうかを調べる。

書　　式——

```
#include <wctype.h>
int iswalpha (wint_t c);
```

解　　説——iswalpha関数は c で指定した文字コードを，現在のロケールのLC_CTYPEカテゴリに照らし合わせ，英字(アルファベット)であるかどうかを調べる。

戻り値——c が英字(アルファベット)ならば0以外の値を返し，異なれば0を返す。

注　　意——通常，iswalpha関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。iswalpha関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はCロケールしかサポートしていない。

規　　格——***Project LIBC Group***，***SYSV***

関連項目——iswalnum，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswupper，iswxdigit

iswascii

用　途——7ビット ASCII 文字かどうかを調べる。

書　式——
```
#include <wctype.h>
int iswascii (wint_t c);
```

解　説——iswascii 関数は c で指定した文字コードが，7ビット ASCII 文字 (0x00 ～ 0x7F) であるかどうかを調べる。

戻り値——c が7ビット ASCII 文字ならば0以外の値を返し，異なれば0を返す。

注　意——通常，iswascii 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswascii 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

iswascii 関数はロケールとは関係がない。

規　格——*Project LIBC Group*，*XPG3*，*AES/OS*，*4.3BSD*，*SYSV*

関連項目——iswalnum，iswalpha，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswupper，iswxdigit

I

iswcntrl

用　　途――制御文字かどうかを調べる。

書　　式――#include <wctype.h>

```
int iswcntrl (wint_t c);
```

解　　説――iswcntrl 関数は *c* で指定した文字コードを，現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**制御文字 (コントロール文字)** であるかどうかを調べる。

戻 り 値――*c* が制御文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意――通常，iswcntrl 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswcntrl 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格――*Project LIBC Group*，*SYSV*

関連項目――iswalnum, iswalpha, iswascii, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit

iswdigit

用　　途——数字かどうかを調べる。

書　　式——#include <wctype.h>

```
int iswdigit (wint_t c);
```

解　　説——iswdigit 関数は *c* で指定した文字コードを，現在のロケールの LC_CTYPE カテゴリに照らし合わせ，数字であるかどうかを調べる。

戻 り 値——*c* が数字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——通常，iswdigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswdigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswascii，iswcntrl，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswupper，iswxdigit

iswgraph

用　　途――表示可能文字かどうかを調べる。

書　　式――#include <wctype.h>

int iswgraph (wint_t c);

解　　説――iswgraph関数は*c*で指定した文字コードを，現在のロケールのLC_CTYPEカテゴリに照らし合わせ，**表示可能文字**(スペースを除く印字可能文字)であるかどうかを調べる。

戻 り 値――*c*が表示可能文字ならば0以外の値を返し，異なれば0を返す。

注　　意――通常，iswgraph関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。iswgraph関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はＣロケールしかサポートしていない。

規　　格――*Project LIBC Group*，*SYSV*

関連項目――iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswlower，iswprint，iswpunct，iswspace，iswupper，iswxdigit

iswlower

用　　途——英小文字かどうかを調べる。

書　　式——

```
#include <wctype.h>
int iswlower (wint_t c);
```

解　　説——`iswlower` 関数は *c* で指定した文字コードを，現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**英小文字**であるかどうかを調べる。

戻 り 値——*c* が英小文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——通常，`iswlower` 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。`iswlower` 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswprint，iswpunct，iswspace，iswupper，iswxdigit

iswprint

用　途——英大文字かどうかを調べる。

書　式——#include <wctype.h>

```
int iswprint (wint_t c);
```

解　説——iswprint 関数は *c* で指定した文字コードを，現在のロケールの LC_CTYPE カテゴリに照らし合わせ，**印字可能文字**であるかどうかを調べる。

戻り値——*c* が印字可能文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，iswprint 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswprint 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswpunct，iswspace，iswupper，iswxdigit

iswpunct

用　　途——記号文字かどうかを調べる。

書　　式——
```
#include <wctype.h>
int iswpunct (wint_t c);
```

解　　説——iswpunct 関数は *c* で指定した文字コードを，現在の**ロケール**の LC_CTYPE カテゴリに照らし合わせ，**記号文字**であるかどうかを調べる。

戻 り 値——*c* が記号文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　　意——通常，iswpunct 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswpunct 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswspace，iswupper，iswxdigit

I

iswspace

用　　途——空白文字かどうかを調べる。

書　　式——

```
#include <wctype.h>
int iswspace (wint_t c);
```

解　　説——iswspace関数はcで指定した文字コードを，現在のロケールのLC_CTYPEカテゴリに照らし合わせ，空白文字(スペース/タブ/改行/復帰/垂直タブ/改頁など)であるかどうかを調べる。

戻り値——cが記号ならば0以外の値を返し，異なれば0を返す。

注　　意——通常，iswspace関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。iswspace関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

*LIBC*はCロケールしかサポートしていない。

規　　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswupper，iswxdigit

iswupper

用　途——英大文字かどうかを調べる。

書　式——#include <wctype.h>

```
int iswupper (wint_t c);
```

解　説——iswupper 関数は *c* で指定した文字コードを，現在の**ロケール**の LC_CTYPE カテゴリに照らし合わせ，**英大文字**であるかどうかを調べる。

戻り値——*c* が英大文字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，iswupper 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。iswupper 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswxdigit

I

iswxdigit

用　途——16 進数字かどうかを調べる。

書　式——#include <wctype.h>
int iswxdigit (wint_t c);

解　説——iswxdigit 関数は *c* で指定した文字コードが，16 進数字であるかどうかを調べる。16 進数字とは 0 ～ 9 と A ～ F，a ～ f のことを指す。

戻り値——*c* が 16 進数字ならば 0 以外の値を返し，異なれば 0 を返す。

注　意——通常，iswxdigit 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__が定義されると実体をもつ関数となる。iswxdigit 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

iswxdigit 関数はロケールとは関係がない。

規　格——*Project LIBC Group*，*SYSV*

関連項目——iswalnum，iswalpha，iswascii，iswcntrl，iswdigit，iswgraph，iswlower，iswprint，iswpunct，iswspace，iswupper

putwc

用　途——ファイルストリームに幅広文字を出力する。

書　式——#include <widec.h>

```
wint_t putwc (wint_t wc, FILE *stream);
```

解　説——putwc 関数は *stream* で指定された**ファイルストリーム**に対して，文字 *wc* を wchar_t 型に変換して出力する。これに応じて，ストリームの**ファイルポインタ**は１幅広文字分進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

戻り値——正常に出力できた場合は *wc* を返し，何らかのエラーによって失敗した場合は，ストリームの**エラー指示子**を設定して WEOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクが一杯である

注　意——putwc 関数はつねに fgetwc 関数の別名である。

規　格——***Project LIBC Group*，*SYSV***

関連項目——fputwc

P

putwchar

用　　途――標準出力ファイルストリームに幅広文字を出力する。

書　　式――#include <widec.h>

wint_t putwchar (wint_t wc);

解　　説――putwchar 関数は標準出力に割り当てられたファイルストリームに対して，*wc*をwchar_t 型に変換して出力する。これに応じて，ストリームのファイルポインタは1幅広文字分進む。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

putwchar 関数は，fputwc 関数に stdout を指定した場合と同じになる。

戻 り 値――正常に出力できた場合は*wc*を返し，何らかのエラーによって失敗した場合は，標準出力ストリームのエラー指示子を設定して WEOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい
- ENOSPC　ディスクが一杯である

注　　意――putwchar 関数はつねにマクロとして定義される。

規　　格――*Project LIBC Group*，*SYSV*

関連項目――fputwc，putwc

putws

用　途——標準出力ファイルストリームに幅広文字列を出力する。

書　式——#include <widec.h>

```
int putws (const wchar_t *s);
```

解　説——putws 関数は**標準出力**に割り当てられた**ファイルストリーム**に対して，*s*が指す幅広文字列と**改行文字**を出力する。幅広文字列は null 文字までとし，最後の null 文字は出力しない。なお，ストリームが追加モードでオープンされている場合，出力はすべてファイルの最後尾に追加される形式で実行される。

putws 関数は fputws 関数とは異なり，幅広文字列の最後に改行文字を付加して出力する。

戻り値——正常に出力できた場合は負ではない値を返し，何らかのエラーによって失敗した場合は，ストリームの**エラー指示子**を設定して EOF を返す。失敗した場合は，変数 errno にその原因を示すエラーコードを設定する。

- EBADF　ファイルストリームに関連するファイルハンドルがおかしい

規　格——*Project LIBC Group*，*SYSV*

関連項目——ferror，fopen，fputws，putwc

P

towlower

用　　途——大文字を小文字に変換する。

書　　式——#include <wctype.h>

```
wint_t towlower (wint_t c);
```

解　　説——towlower 関数は *c* で指定された文字を，現在のロケールの LC_CTYPE カテゴリにしたがって大文字から小文字に変換する。ただし，変換は大文字に対してのみ行われ，それ以外の文字は変化しない。

戻 り 値——*c* を小文字に変換した文字を返す。

注　　意——通常，towlower 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。towlower 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC はＣロケールしかサポートしていない。

規　　格——***Project LIBC Group，SYSV***

関連項目——towupper

towupper

用　途――小文字を大文字に変換する。

書　式――#include <wctype.h>

```
wint_t towupper (wint_t c);
```

解　説――towupper 関数は *c* で指定された文字を，現在のロケールの LC_CTYPE カテゴリにしたがって小文字から大文字に変換する。ただし，変換は小文字に対してのみ行われ，それ以外の文字は変化しない。

戻り値――*c* を大文字に変換した文字を返す。

注　意――通常，towupper 関数はマクロとして定義されるが，__NO_CTYPE_INLINE__ が定義されると実体をもつ関数となる。towupper 関数がマクロとして展開される場合は，引数が副作用を伴わないように注意すること。

LIBC は C ロケールしかサポートしていない。

規　格――***Project LIBC Group*，*SYSV***

関連項目――towlower

T

ungetwc

用　　途――入力ファイルストリームに幅広文字を押し戻す。

書　　式――#include <widec.h>

int ungetwc (wchar_t wc, FILE *stream);

解　　説――ungetwc 関数は，*stream* で指定された入力用のファイルストリームに *wc* を押し戻す。ただし，*wc* に EOF を指定した場合は何も行わない。

押し戻されたデータは後に続く読み出し操作で最優先に取り出されることになるが，あくまでもバッファ上の操作であり，ファイルの実体は一切変更されない。また押し戻しが行われた後は，ストリームに設定されていた終端指示子はクリアされる。

戻り値――正常に押し戻せた場合は *wc* を int 型に変換して返し，失敗した場合は EOF を返す。

注　　意――ungetwc 関数は最低限1幅広文字を押し戻せることを保証しているが，それ以上については，状況によっては押し戻せないことがあるので注意すること。

規　　格――*Project LIBC Group*，*SYSV*

関連項目――fseek，fsetpos，getwc，rewind，setbuf，ungetwc

wcscat

用　　途——文字列をほかの文字列に連結する。

書　　式——#include <wstring.h>

```
wchar_t *wcscat (wchar_t *string1, const wchar_t *string2);
```

解　　説——wcscat 関数は，*string2*の指す文字列を *string1*の指す文字列の最後にコピーし，連結した文字列の最後に null 幅広文字をつける。*string1*の末尾の null 幅広文字には，*string2*の最初の文字を重ねてコピーする。

戻 り 値——*string1*へのポインタを返す。

注　　意——*string1*は，連結した結果の文字列を格納するに十分な領域を指していなければならない。1 文字は 1 幅広文字を意味する。

規　　格——*MSE*

関連項目——wcsncat

wcschr

用　途——文字列中から指定文字を検索する。

書　式——#include <wstring.h>

```
wchar_t *wcschr (const wchar_t *string, wint_t character);
```

解　説——wcschr 関数は *string* が指す文字列中から，*character* で指定された文字を検索する。*string* 末尾の null 幅広文字も検索の対象となり得る。*character* は，内部で wint_t 型から wchar_t 型に変換される。

戻り値——*string* 中で最初に現れた *character* へのポインタを返す。見つからない場合は NULL を返す。

注　意——1文字は1幅広文字を意味する。

規　格——*MSE*

関連項目——wcscspn，wcspbrk，wcsrchr，wcsspn，wcstok

wcscmp

用　途——2つの文字列を比較する。

書　式——#include <wstring.h>

```
int wcscmp (const wchar_t *string1, const wchar_t *string2);
```

解　説——wcscmp 関数は，*string1* の指す文字列と *string2* の指す文字列を比較する。比較は，null 幅広文字が検出された時点で終了する。

戻り値——比較の結果，2つの文字列がまったく同じならば0を返す。異なる場合，その位置での *string1* 側の幅広文字が *string2* 側の幅広文字よりも大きければ正の値を，小さければ負の値を返す。

注　意——1文字は1幅広文字を意味する。

規　格——*MSE*

関連項目——wcsncmp

W

wcscoll

用　　途――2つの幅広文字列をロケールを用いて比較する。

書　　式――#include <wstring.h>

```
int wcscoll (const wchar_t *string1, const wchar_t *string2);
```

解　　説――wcscoll関数は，*string1*の指す幅広文字列と*string2*の指す幅広文字列を現在のロケールのLC_COLLATEカテゴリを基に比較する。比較は，null幅広文字が検出された時点で終了する。

戻 り 値――比較の結果，2つの幅広文字列がまったく同じならば0を返す。異なる場合，その位置での*string1*側の幅広文字が*string2*側の幅広文字よりも大きければ正の値を，小さければ負の値を返す。

注　　意――*LIBC*はCロケールしかサポートしていないので，strcmp関数と同じである。1幅広文字は1バイトを意味する。

規　　格――***Project LIBC Group***，***MSE***

関連項目――wcscmp，wcsxfrm

wcscpy

用　途――文字列をコピーする。

書　式――#include <wstring.h>

```
wchar_t *wcscpy (wchar_t *string1, const wchar_t *string2);
```

解　説――wcscpy 関数は *string2* の指す文字列を，null 幅広文字を含めて *string1* の指す領域にコピーする。

戻り値――*string1* へのポインタを返す。

注　意――領域が重なっていた場合の動作は未定義である。また *string1* は，*string2* の指す文字列を格納するのに十分な領域を指していなければならない。1 文字は 1 幅広文字を意味する。

規　格――*MSE*

関連項目――wcsncpy

wcscspn

用　　途——指定文字列に含まれない文字が，ほかの文字列の先頭から何文字続いているかを調べる。

書　　式——
```
#include <wstring.h>
size_t wcscspn (const wchar_t *string1, const wchar_t *string2);
```

解　　説——wcscspn関数は*string1*の指す文字列と*string2*の指す文字列を比較し，*string1*の先頭から*string2*に含まれない文字で構成されている部分の長さを調べる。

戻り値——*string1*の先頭から，*string2*に含まれない文字で構成されている部分の長さを返す。

注　　意——1文字は1幅広文字を意味する。

規　　格——*MSE*

関連項目——wcschr，wcspbrk，wcsrchr，wcsspn，wcstok

wcsdup

用　　途――新しい領域を確保して文字列をコピーする。

書　　式――#include <wstring.h>

```
wchar_t *wcsdup (const wchar_t *string);
```

解　　説――wcsdup 関数は *string* の指す文字列を，null 幅広文字を含めて malloc 関数によって確保した領域にコピーする。

戻 り 値――正常にコピーできた場合はコピー先へのポインタを返し，失敗した場合は NULL を返して，変数 errno にその原因を示すエラーコードを設定する。

- ENOMEM　メモリが足りなくなった

注　　意――1 文字は 1 幅広文字を意味する。

規　　格――*Project LIBC Group*

関連項目――malloc，wcscpy

wcslen

用　　途——文字列の長さを調べる。

書　　式——#include <wstring.h>

```
size_t wcslen (const wchar_t *string);
```

解　　説——wcslen関数は，*string*の指す文字列末尾のnull幅広文字を除いた文字列の長さを調べる。

戻り値——末尾のnull幅広文字を除いた文字列の長さを返す。

注　　意——1文字は1幅広文字を意味する。

規　　格——*MSE*

関連項目——wcscpy

wcsncat

用　　途――文字列を指定文字数だけほかの文字列に追加する。

書　　式――#include <wstring.h>

```
wchar_t *wcsncat (wchar_t *string1, const wchar_t *string2,
                  size_t n);
```

解　　説――wcsncat 関数は，*string2*の指す文字列を *string1* の指す文字列の最後にコピーし，連結した文字列の最後に null 幅広文字をつける。*string1* の末尾の null 幅広文字には，*string2*の最初の文字を重ねてコピーする。コピーは null 幅広文字が検出されるか，*n*文字コピーした時点で終了する。

戻 り 値――*string1* へのポインタを返す。

注　　意――*string1* は，連結した結果の文字列を格納するのに十分な領域を指していなければならない。1 文字は 1 幅広文字を意味する。

規　　格――*MSE*

関連項目――wcscat

wcsncmp

用　途——2つの文字列を指定文字数だけ比較する。

書　式——

```
#include <wstring.h>
int wcsncmp (const wchar_t *string1, const wchar_t *string2,
                        size_t n);
```

解　説——wcsncmp 関数は，*string1* の指す文字列と *string2* の指す文字列を比較する。比較は null 幅広文字が検出されるか，*n* 文字比較した時点で終了する。

戻り値——比較の結果，2つの文字列がまったく同じならば0を返す。異なるか *n* 文字を比較し終えた場合，その位置での *string1* 側の文字が *string2* 側の文字よりも大きければ正の値を，小さければ負の値を返す。

注　意——1文字は1幅広文字を意味する。

規　格——***MSE***

関連項目——wcscmp

wcsncpy

用　途——文字列を指定文字数だけコピーする。

書　式——#include <wstring.h>

```
wchar_t *wcsncpy (wchar_t *string1, const wchar_t *string2,
        size_t n);
```

解　説——wcsncpy 関数は *string2* の指す文字列の最初の *n* 文字までを，*string1* の指す領域にコピーする。コピーは null 幅広文字が検出されるか，*n* 文字コピーした時点で終了する。

戻り値——*string1* へのポインタを返す。

注　意——領域が重なっていた場合の動作は未定義である。また *string1* は，*n* 文字を格納するのに十分な領域を指していなければならない。1 文字は 1 幅広文字を意味する。

規　格——*MSE*

関連項目——wcscpy

wcspbrk

用　途——指定文字列に含まれる文字がほかの文字列に存在するかどうかを調べる。

書　式——

```
#include <wstring.h>
wchar_t *wcspbrk (const wchar_t *string1,
                  const wchar_t *string2);
```

解　説——wcspbrk 関数は，*string1* の指す文字列と *string2* の指す文字列を比較し，*string2* に含まれる文字が最初に現れる場所を検索する。

戻り値——*string1* 中で，*string2* に含まれる文字が最初に現れる位置へのポインタを返す。見つからない場合は NULL を返す。

注　意——1 文字は 1 幅広文字を意味する。

規　格——*MSE*

関連項目——wcschr，wcscspn，wcsrchr，wcsspn，wcstok

wcsrchr

用　途——文字列中から指定文字が最後に現れる位置を検索する。

書　式——#include <wstring.h>

```
wchar_t *wcsrchr (const wchar_t *string, wint_t character);
```

解　説——wcsrchr 関数は *string* の指す文字列中から，*character* で指定された文字が最後に現れる位置を検索する。*string* 末尾の null 幅広文字も検索の対象となり得る。*character* は，内部で wint_t 型から wchar_t 型に変換される。

戻り値——*string* 中で最後に現れる *character* へのポインタを返す。見つからない場合は NULL を返す。

注　意——1 文字は 1 幅広文字を意味する。

規　格——*MSE*

関連項目——wcschr, wcscspn, wcspbrk, wcsspn, wcstok

wcsspn

用　　途——指定文字列に含まれる文字が，ほかの文字列の先頭から何文字続いているかを調べる。

書　　式——#include <wstring.h>

```
size_t wcsspn (const wchar_t *string1, const wchar_t *string2);
```

解　　説——wcsspn 関数は，*string1* の指す文字列と *string2* の指す文字列を比較し，*string1* の先頭から *string2* に含まれる文字で構成されている部分の長さを調べる。

戻り値——*string1* の先頭から，*string2* に含まれる文字で構成されている部分の長さを返す。

注　　意——1文字は1幅広文字を意味する。

規　　格——*MSE*

関連項目——wcschr，wcscspn，wcspbrk，wcsrchr，wcstok

wcstod

用　　途——幅広文字列を double 型倍精度浮動小数に変換する。

書　　式——

```
#include <wstring.h>
double wcstod (const wchar_t *nptr, wchar_t **endptr);
```

解　　説——wcstod 関数は *nptr* で指定された幅広文字列を，まず 3 つの部分文字列すなわち先頭の空白部分，浮動小数部分，認識不能部分 (終端の null 幅広文字を含む) に分割する。wcstod 関数はこれらの部分文字列のうち，浮動小数部分の文字列を double 型倍精度浮動小数に変換し，その値を返す。

この浮動小数部分は符号 (省略可能)，浮動小数および指数 (省略可能) からなり，指数部分は "e" か "E" に続いて符号 (省略可能) と指数値からなる。

-3.1415926，1.8794E+15，2.42e-6

もし *endptr* が NULL でない場合は，*endptr* に最後の認識不能部分へのポインタを格納する。この部分は終端の null 文字を含むため，1 文字以上になる。

小数点として認識する文字は，ロケールの LC_NUMERIC カテゴリに影響される。

戻 り 値——変換した結果を返す。ただし結果がオーバーフローした場合は HUGE_VAL を返し，アンダーフローした場合は 0 を返して，変数 errno にその原因を示すエラーコードを設定する。

nptr が空文字であったり，すべて空白だったり，認識不能文字だったりして浮動小数部分の文字列が見つからない場合は変換は行われず，0 を返す。また，このとき *endptr* には *nptr* の値が格納される。

- ERANGE　オーバーフローあるいはアンダーフローを起こした

注　　意——*LIBC* は C ロケールしかサポートしていない。

規　　格——*Project LIBC Group*，*MSE*

関連項目——wcstol，wcstoul

wcstok

用　　途――文字列を指定した区切り文字でトークンに分ける。

書　　式――#include <wstring.h>

```
wchar_t *wcstok (wchar_t *string, const wchar_t *delim);
```

解　　説――wcstok関数を連続して呼ぶことにより，*string*が指す文字列を語句(トークン)に分解することができる。各語句は，*delim*が指す文字列に含まれる区切り文字(デリミタ)によって区切られる。

最初にwcstok関数を呼び出すときは，デリミタ文字列を*string*に指定する。以降，wcstok関数を呼び出すときには，*string*にNULLを指定する。デリミタ文字列は，wcstok関数を呼び出すたびに変えることが可能。*string*にデリミタ文字列が続く場合は無視される。

戻り値――語句が見つかれば，最後の区切り記号の位置にnull幅広文字を設定し，語句の最初の文字へのポインタを返す。見つからなければNULLを返す。

注　　意――wcstok関数は与えられた*string*の領域を書き換えるので，元のデータが必要な場合は自分で保存しておく必要がある。1文字は1幅広文字を意味する。

規　　格――*MSE*

関連項目――wcschr，wcscspn，wcspbrk，wcsrchr，wcsspn，wcstok

wcstol

用　　途——幅広文字列をlong型整数に変換する。

書　　式——#include <wstring.h>

```
long wcstol (const wchar_t *nptr, wchar_t **endptr, int radix);
```

解　　説——wcstol関数は*nptr*で指定された幅広文字列を，まず3つの部分文字列すなわち先頭の空白部分，整定数部分，認識不能部分(終端のnull幅広文字を含む)に分割する。wcstol関数はこれらの部分文字列のうち，整定数部分の文字列を，*radix*で指定した値を基数としてlong型整数に変換し，その値を返す。

整定数部分は符号(省略可能)から始まり，基数が1～10ならば0～9の文字を，また基数が10～36ならばこれに加えてA～Zおよびa～zの文字が数値として認識される。

もし基数*radix*が0の場合，入力文字列の基数は8，10，16のいずれかであり，文字列の特徴から動的に決定される。すなわち01～07から始まれば8，0xから始まれば16，それ以外ならば10である。また基数が16の場合は，接頭辞0xも数値の一部として認識される。

ただし，数値の最後のu，U，l，Lなどのunsigned型指定文字，long型指定文字は数値としては認識されないので注意すること。

*endptr*がNULLでない場合は，*endptr*に最後の認識不能部分へのポインタを格納する。この部分は終端のnull文字を含むため，1文字以上になる。

戻 り 値——変換した結果を返す。ただし結果がlong型で表現しきれない場合は，符号に応じてLONG_MAXあるいはLONG_MINを返し，変数errnoにその原因を示すエラーコードを設定する。

*nptr*が空文字だったり，すべて空白だったり，認識不能文字だったりして整定数部分の文字列が見つからない場合，変換は行われず0を返す。また，このとき*endptr*には*nptr*の値が格納される。

- ERANGE　変換結果がlong型で表現することができない

W

注　　意——*LIBC*はCロケールしかサポートしていない。

規　　格——***Project LIBC Group*，*MSE***

関連項目——wcstod，wcstoul

wcstoul

用　　途——幅広文字列を unsigned long 型整数に変換する。

書　　式——#include <wstring.h>

```
unsigned long wcstoul (const wchar_t *nptr, wchar_t **endptr,
                        int radix);
```

解　　説——wcstoul 関数は *nptr* で指定された幅広文字列を，まず3つの部分文字列すなわち先頭の空白部分，整定数部分，認識不能部分 (終端の null 幅広文字を含む) に分割する。wcstoul 関数はこれらの部分文字列のうち，整定数部分の文字列を，*radix* で指定した値を基数として unsigned long 型整数に変換し，その値を返す。

整定数部分は符号 (省略可能) から始まり，基数が 1 ～ 10 ならば 0 ～ 9 の文字を，また基数が 10 ～ 36 ならばこれに加えて A ～ Z および a ～ z の文字が数値として認識される。

もし基数 *radix* が 0 の場合は，入力文字列の基数は 8，10，16 のいずれかであり，文字列の特徴から動的に決定される。すなわち 01 ～ 07 から始まれば 8，0x から始まれば 16，それ以外ならば 10 である。また基数が 16 の場合は，接頭辞 0x も数値の一部として認識される。

ただし，数値の最後の u，U，l，L などの unsigned 型指定文字，long 型指定文字および負の符号 “-” は数値としては認識されないので注意すること。

endptr が NULL でない場合は，*endptr* に最後の認識不能部分へのポインタを格納する。この部分は終端の null 文字を含むため，1 文字以上になる。

戻 り 値——変換した結果を返す。ただし結果が unsigned long 型で表現しきれない場合は ULONG_MAX を返し，変数 errno にその原因を示すエラーコードを設定する。

nptr が空文字だったり，すべて空白だったり，認識不能文字だったりして整定数部分の文字列が見つからない場合，変換は行われず 0 を返す。また，このとき *endptr* には *nptr* の値が格納される。

- ERANGE　変換結果が unsigned long 型で表現することができない

注　　意——*LIBC* は C ロケールしかサポートしていない。

規　　格——***Project LIBC Group*，*MSE***

関連項目——wcstod，wcstol

wcswcs

用　途——指定文字列がほかの文字列に存在するかどうかを調べる。

書　式——#include <wstring.h>

```
size_t wcswcs (const wchar_t *string1, const wchar_t *string2);
```

解　説——wcswcs 関数は *string1* の指す文字列と *string2* の指す文字列とを比較し，*string1* 中で，文字列 *string2* が最初に現れる位置を検索する。

戻り値——*string1* 中で，*string2* が最初に現れる位置へのポインタを返す。

注　意——1文字は1幅広文字を意味する。

規　格——*MSE*

関連項目——wcschr, wcspbrk, wcsrchr, wcsspn, wcstok

wcsxfrm

用　途——2つの幅広文字列をロケールを用いて指定の幅広文字数だけコピーする。

書　式——#include <wstring.h>

```
size_t wcsxfrm (wchar_t *string1, const wchar_t *string2,
                size_t n);
```

解　説——wcsxfrm関数は，*string2*が指す幅広文字列の最初から*n*幅広文字を現在のロケールのLC_COLLATEカテゴリを基に変換し，*string1*の指す領域にコピーする。コピーはnull幅広文字が検出されるか，*n*幅広文字コピーした時点で終了する。*string2*末尾のnull幅広文字も変換の対象となり得る。また*n*が0の場合，*string1*はNULLでもよい。

戻り値——変換された幅広文字の文字数(null幅広文字は含まない)を返すが，*n*が0の場合は単に*string2*の長さを返す。

注　意——領域が重なっていた場合の動作は未定義である。また*string1*は，*string2*を変換した幅広文字列を格納するのに十分な領域を指していなければならない。

*LIBC*はCロケールしかサポートしていないので，基本的にはstrncpy関数と同じである。1文字は1バイトを意味する。

規　格——***Project LIBC Group，MSE***

関連項目——wcscoll，wcsncpy

Chapter 7
Appendix A

A……………………………………… C 標準関数

❖ 数値演算

❖ プロセスの環境設定・取得

❖ コンソール直接入出力

❖ 文字の判定と変換

❖ 低水準ファイル入出力とファイル名操作

❖ 割り込み処理

❖ 大域ジャンプ

❖ メモリ管理

❖ プロセス操作

❖シグナル操作

❖ソート

❖標準ファイル入出力

❖ 文字列とメモリ領域の操作

❖ 時間の取得と設定

❖ ユーザデバイス管理

❖ その他

B……………………DOSコール

C……IOCSコール

D ……………………………… マルチバイト文字

E ……………………………………………SCSI コール

F ……………… 幅広文字

索 引

Symbols

A

B

C

E

F

J

へ

ほ

ま

み

む

め

も

ゆ

よ

ら

り

る

れ

ろ

わ

SOFT BANK ソフトバンク

ISBN4-89052-534-3

C0055 P6300E

9784890525348

2冊セット定価6,300円
(セット本体6,117円　分売不可)

1910055063006

X68k
Programming Series
(#2)
X680x0
libc